1990-12

Implementation of residue code as a design for testability strategy using GENESIL Silicon Compiler

Lawson, John Ernest

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF RESIDUE CODE AS A DESIGN FOR TESTABILITY STRATEGY USING GENESIL SILICON COMPILER

by

John Ernest Lawson

December, 1990

Thesis Advisor: Chyan Yang
Co-Advisor: Herschel H. Loomis, Jr.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) EC	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
IMPLEMENTATION OF RESIDUE CODE AS A DESIGN FOR TESTABILITY STRATEGY USING GENESIL SILICON COMPILER

12. PERSONAL AUTHOR(S)
LAWSON, John E.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	1990 December	111

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	design for testability; scan design; built-in self-test; residue code; notch filter; silicon compiler

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis describes the need for including design for testability in a VLSI chip design and provides information on implementing a DFT strategy using the GENESIL Silicon compiler. Two structured techniques of design for testability, Scan Design and Built-in Self Test, are discussed. Also, the methodology used to implement the residue code with GENESIL for testing the multiply-add module of a second-order Infinite Impulse Response notch filter is presented. The cost, in terms of increased hardware and decreased performance, associated with implementing the residue code is examined by comparing modulo-3 and modulo-15 checking algorithms.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL YANG, Chyan	22b. TELEPHONE (Include Area Code) 408-646-2266 22c. OFFICE SYMBOL EC/Ya

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603 UNCLASSIFIED

Approved for public release; distribution is unlimited.

IMPLEMENTATION OF RESIDUE CODE AS A DESIGN FOR TESTABILITY STRATEGY USING GENESIL SILICON COMPILER

by

John Ernest Lawson
Lieutenant, United States Navy
B.S., University of Mississippi, 1983

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1990

Author: [redacted]
John Ernest Lawson

Approved by: [redacted]
Chyan Yang, Thesis Advisor

[redacted]
Herschel H. Loomis, Jr., Co-Advisor

[redacted]
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

This thesis describes the need for including design for testability in a VLSI chip design and provides information on implementing a DFT strategy using the GENESIL Silicon Compiler. Two structured techniques of design for testability, Scan Design and Built-in Self Test, are discussed. Also, the methodology used to implement the residue code with GENESIL for testing the multiply-add module of a second-order Infinite Impulse Response notch filter is presented. The cost, in terms of increased hardware and decreased performance, associated with implementing the residue code is examined by comparing modulo-3 and modulo-15 checking algorithms.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

Barry Johnson [Ref. 1:p. 7] defines a **test** as a means by which the existence and quality of certain attributes of an electronic system can be determined. For instance, if a computer is advertised to execute one million instructions per second, you would want to design a test to verify that the computer performs at that particular rate. Furthermore, **testability** is the ability to test for specific attributes within an electronic system [Ref. 1:p. 8].

Electronic systems, such as digital computers, have become so common and useful in modern society that they are indispensable. Their rapid advances have been made possible by the dramatic progress toward Very Large Scale Integration (VLSI) in the semiconductor circuit technologies achieved in recent years. The continually growing significance and complexity of today's electronic systems demands that special features be incorporated into the system to support testing in a simple and straightforward manner. **Design for testability** (DFT) is the process by which such features are included.

Integration means realization and packaging of multiple circuits in a single "smallest unit of fabrication" which, in semiconductor technologies, is called a chip [Ref. 2:p. 6].

The importance of circuit integration lies in its inherent capabilities for reducing the cost of the electronic circuits' fabrication as well as improving their performance and reliability. It reduces costs by packaging more circuits in each unit of fabrication and allowing much of the production processes to be automated; it improves the circuits' performance by decreasing their dimensions and the signal propagation delays, thus, increasing their operational speeds; it improves their reliability by using fewer solder joints and shortening interconnections [Ref. 2:p. 7].

Advances in circuit integration have been impressive and are expected to continue at an accelerated pace. However, as VLSI circuit densities increase, it is generally recognized that the problems in testing become correspondingly complex and difficult in at least two ways. First, circuits to be tested have become so complicated that they can no longer be handled by one person. This has lead to difficulties in the planning and design for testing. The use of computer-aided design (CAD) tools is one way to overcome these problems. Second, VLSI circuits have become so fast, compact and inaccessible that conventional methods of testing are no longer adequate. To cope with these difficulties, more and better use of computers to help manage complex tasks has led to the use of automatic test equipment (ATE) [Ref. 2:p. 41].

Future advances in VLSI circuit technologies will further advance integration and speeds. Thus, circuit testing will

become more complex, difficult and costly, both in test execution and in test equipment - unless something is done to reverse this trend. In fact, there are cases in which integrated circuits and systems were designed and built but were not fabricated as products because they turned out to be too costly or even "impossible" to test [Ref. 2:p. 14].

It should now be clear that although the per-chip fabrication and assembly costs have decreased with ever improving technology, the per-chip testing costs have increased as a percentage of the total chip cost, and this escalation in testing costs and testing difficulties can seriously slow or stop the on going development for larger and more complicated electronic systems [Ref. 2:p. 15]. Therefore, the need for ccnsidering design for testability techniques during chip design can largely be predicted on one factor: cost [Ref. 3:p. 100]. The inclusion of testability design from the beginning of a project can make testing more economical and effective.

Prior to circuit integration, there was no requirement in the design of electronic circuits and systems for testability. Circuits and systems were built using large-sized discrete components (vacuum tubes, resistors, capacitors, transistors, etc.) mounted on cards or boards with most node points accessible to direct probing for testing. The behavior of these circuits could be determined by monitoring the voltages at the various node points, and these circuits were considered

testable. However, smaller and faster electronic circuits were developed, and packaging densities were continually increased until circuit components became too crowded to be conveniently accessible by probing. In order to sustain accessibility, "test points" were eventually added - at the expense of either spacing out the components or providing extra peripheral area contacts [Ref. 2:p. 39].

Following circuit integration, the dimensions of circuit components were drastically reduced further. As a result, most VLSI circuit components have lost their individual accessibility. It is basically the increasing inaccessibility of VLSI circuits (single circuits as well as groups of circuits) that is producing difficulties in testing. One reason for the increased inaccessibility is that as the number of circuits on a VLSI chip grows they require more input/output (I/O) pins for normal system operation, but due to requirements for making reliable solder joints, the miniaturization of the I/O pads on the chip has not kept pace with the growing number of transistors within the chip. Therefore, the relative number of I/O pins available for direct probing or testing has decreased. Also, it becomes increasingly difficult to feed stimulus and response signals at high speeds through continually decreasing dimensions of solder pads, connectors, fixtures and probes without some noise and signal distortion that might affect the reliability of a test. Thus, it can be seen that it is not feasible to

provide indefinitely more I/O connections without degrading their quality for signal communications [Ref. 2:p. 59].

As systems grew and became more complicated, they had to be subdivided into parts that were built and tested separately. Following individual testing, the parts were assembled into a system and then tested as a whole for correct functioning. This method is basically the "conventional" way of testing [Ref. 2:p. 38]. Conventional test methods of VLSI circuits are faced with ever increasing and insurmountable difficulties due to technological barriers. Attempts to match the ongoing advances in circuit integration with further mechanical miniaturization are destined to be dead-end: each incremental step in the dimensional reduction can be accomplished only against escalated technological difficulties and at disproportionately increased costs. Conventional testing relies primarily on adding improved mechanical means and not on incorporation and use of additional circuits in the object to be tested for the purpose of facilitating its testing. However, design for testability is an integral part of the circuit and system design, and can be considered to be electronic in nature vice mechanical [Ref. 2:p. 48].

Common characteristics of conventional test methods are:

1. Conventional methods cannot test in-system because when a part is in the system its inputs and outputs are connected to some other parts. The part must be tested

in isolation which means its connections will need to be severed either by using electronic switching or by physically detaching the part from the system. Since conventional methods do not rely on the incorporation and use of the additional circuits required for electronic switching, they are unable to test parts in-system.

2. Conventional methods rely on test equipment to generate test patterns for the system-part being tested outside of system and to capture its output response. Thus, conventional test methods rely on Direct Signal Feeding which is the feeding of signals directly through the test-interface during testing.
3. Conventional methods require timing controls that are generated and driven by test equipment that is external to and not considered part of the system. This reliance on the use of tester-driven timing is because the part to be tested does not usually have all the needed timing controls when it is outside of and separated from the system.

Some of the consequences of conventional testing are that system-parts tested outside-of-system often introduce unavoidable uncertainties because it is virtually impossible to reproduce exactly the systems-operation environment in a test-fixture setup. In practice, faults which are detected in a system often cannot be reproduced any more in the

disassembled parts. With system-parts designed for in-system testability, testing will, therefore, be more efficient and cost effective, and the insufficiency of testing system-parts outside of and separate from the system can be eliminated.

Design for testability is one possible approach to overcome the technological barriers of conventional testing and increase individual circuit component accessibility. The goal of DFT is to find ways to make all parts including the assembled system easier, more efficient and less costly to test in-system despite increasing inaccessibility of circuits and a shortage of I/O pins for test purposes. Thus, DFT must achieve some "sufficient" degree of testability by using only a "small" number of extra I/O pins for test purposes, and it must achieve this result at the cost of only a "small" amount of hardware overhead and performance penalty [Ref. 2:p. 83].

Design for testability requires the incorporation of additional circuits through careful design in order to provide controllability and observability of the system. **Controllability** refers to the degree to which a node internal to a circuit can be set to a given logic level [Ref. 4:p. 97]. On the other hand, **observability** can be defined as the ability to observe the logic level of a given internal node at the output of the design [Ref. 4:p. 97]. One of the chief aims of design for testability is to find new ways to control and observe a large number of nodes within the system to determine with a high degree of confidence if the system is "fault free."

If a circuit demonstrates failure which causes deviations from the specified performance behavior it is said to contain **faults** [Ref. 5:p. 1]. The two major categories of faults are physical circuit defect faults and design faults. Physical failures are a result of manufacturing defects or wear-out in the field. Some examples of manufacturing defects include faulty transistors, open contacts, electrical shorts between circuit parts and broken lines [Ref. 5:p. 1]. Major contributors to these physical defects are lithographic errors during the manufacture of VLSI circuits such as alignment failures and mask errors [Ref. 6:p. 693]. Improper handling of delicate electronic circuits can lead to input gate breakdown due to static electricity, and the intrusion of moisture during the packaging of integrated circuits can lead to failure. Wear-out or long term failures are caused by aluminum metal corrosion or high current densities in thin wires that can result in metal migration [Ref. 6:p. 695]. Design faults are caused by improper VLSI circuit connections due to either design mistakes or implementation mistakes.

Fault models are used to describe the effect of a physical failure on the performance of the system and can include modeling faults down to the individual transistor circuit level, but usually fault models only consider faults down to the logic circuit level (also called gate level by some). The reason why a logic circuit level fault model is most often used is that this model can represent faults for many

different technologies. An example of a logic circuit level model that is technology dependent is the **logical stuck-fault model** [Ref. 1:p. 32].

The logical stuck-fault model is often referred to as the stuck-at-0 (s-a-0), stuck-at-1 (s-a-1) fault model or simply the stuck-fault model, and it is a representation that assumes all faults will appear as lines in the logic diagram being physically stuck at a logic 1 or logic 0 value [Ref. 1:p. 42].

Three basic assumptions of the stuck-fault model are:

1. a fault results in a model responding as if one of its inputs or outputs is physically stuck at logic 1 or 0
2. the circuits basic functionality is not altered by the fault
3. the fault is permanent

The logic module can either be a single gate or a collection of gates that implements some logic function [Ref. 1:p. 32].

The CMOS inverter of Figure 1.1 shows how a stuck-fault can occur. If the input line is shorted to ground (logic 0) at point A then the gate output will be stuck-at-1 (s-a-1), regardless of what the inverter input is. If the line is broken at point B then the gate will produce the expected output of logic 1 when a logic 0 is input and the *p*-channel transistor turns on. However, if the input is a logic 1 the *p*-channel transistor will turn off, but the *n*-channel

transistor will never turn on because the line is broken. As a result, the output will remain at a logic 1 for a period of time dependent on leakage currents. If a high speed stream of data consisting of both logic 1's and 0's is being input to a device which includes the inverter, the output may appear as a permanent s-a-1 fault [Ref. 7:p. 7].

The AND gate of Figure 1.2 is used to show how to test for stuck-faults. For example, if this AND gate has a stuck-at-0 fault on input line A, the gate will always produce a logic 0 at the output. The applied input, however, is free to assume any value. Input pairs of (0,0), (0,1) or (1,0) on lines A, B will all produce correct outputs on line F. However, when input (1,1) is used the output should be logic 1 but will instead be logic 0 due to the s-a-0 fault on input line A.

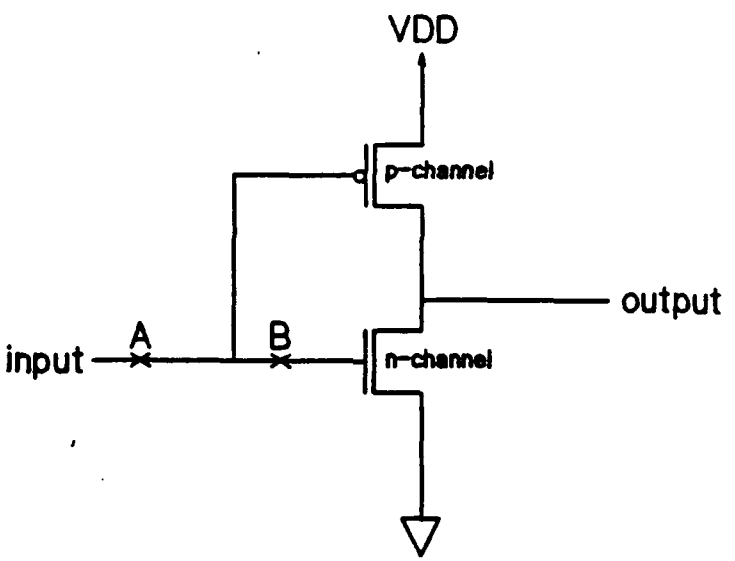


Figure 1.1 CMOS Inverter stuck-fault model [Ref. 7]

The input pattern (1,1) is, therefore, a test of s-a-0 fault. Most applications of the stuck-fault model limit the number of faults that can occur at any one time. It is typically assumed that a circuit will never have more than one stuck-fault. The single fault assumption is commonly used to simplify the process of analyzing a circuit or generating test patterns. In a logic gate that contains *n* lines, a possibility of at most 2*n* unique, single, stuck faults can occur because each line can exhibit a s-a-0 or s-a-1 fault [Ref. 1:p. 33].

The design of test vector inputs capable of detecting these 2*n* stuck-faults is one of the problems in testing. The AND gate of Figure 1.3 illustrates that certain input patterns can determine the presence of more than one stuck-fault. For example, an input pattern of (1, 1, 0, 1) that produces an

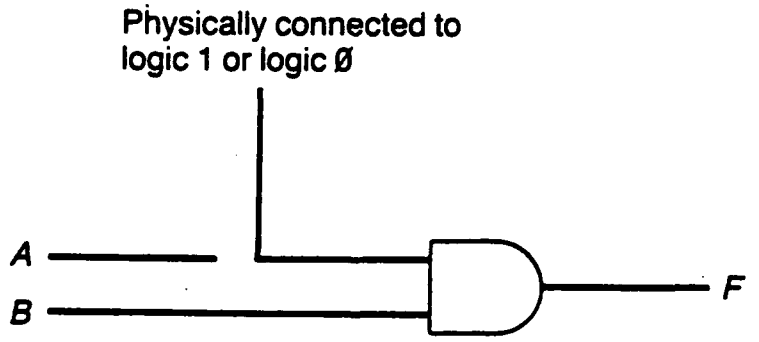
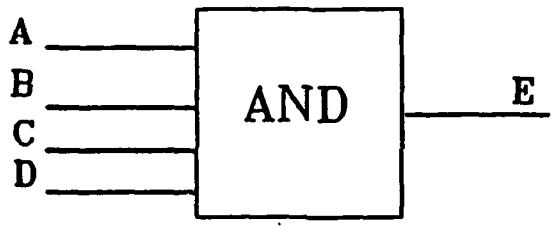


Figure 1.2 AND gate stuck-fault testing [Ref. 1]



Input Faults		Pattern to Detect Fault A B C D
A	S-A-0	1 1 1 1
A	S-A-1	0 1 1 1
B	S-A-0	1 1 1 1
B	S-A-1	1 0 1 1
C	S-A-0	1 1 1 1
C	S-A-1	1 1 0 1
D	S-A-0	1 1 1 1
D	S-A-1	1 1 1 0

Output Faults		Pattern to Detect Fault A B C D
E	S-A-0	1 1 1 1
E	S-A-1	any input combination that contains one or more 0 inputs

Patterns Needed to Completely Test Gate

A	B	C	D
1	1	1	1
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

Figure 1.3 Stuck-fault vectors for fault detection [Ref. 8]

output of logic 1 means that either input C is s-a-1 or output E is s-a-1 [Ref. 8:p. 7, 8]. Furthermore, Figure 1.3 shows that only five test vectors are needed to completely test the functionality of the gate.

One method often used to quantify the effectiveness of a fault model is a coverage parameter. If the actual physical fault is accurately represented by the chosen fault model then a fault model is said to cover a fault. Ideally, a fault model should cover 100 percent of all physical faults, but this is seldom the case.

The classic example of a stuck-fault model that does not cover a very specific and practical physical fault is the CMOS NOR gate [Ref. 9] shown in Figure 1.4. The circuit is a combination of two *p*-channel transistors in series with two parallel *n*-channel transistors. A path for current flow is set-up from either *VDD* or *VSS* to the circuit output based on the input values, A and B. If both A and B are at logic 0, both *p*-channel transistors are conducting while both *n*-channel transistors are off. The output is forced to a logic 1 by the path established between *VDD* and the output. Similarly, if either or both inputs are logic 1, the corresponding *p*-channel transistors are turned off while one or both *n*-channel transistors are conducting. Thus, a path from the output to *VSS* is set up, forcing the output to a logic 0 [Ref. 1:p. 33].

The NOR circuit can easily assume many faults that behave as stuck-faults [Ref. 1:p. 33]. One possible example is when

the drain and source of one of the *n*-channel transistors become shorted together, causing the device to always output a logic 0. As a result, the fault can be modeled as the output s-a-0. Another example is if input line A becomes shorted to *VDD* the output of the circuit will always be a logic 0, behaving as if input line A is s-a-1.

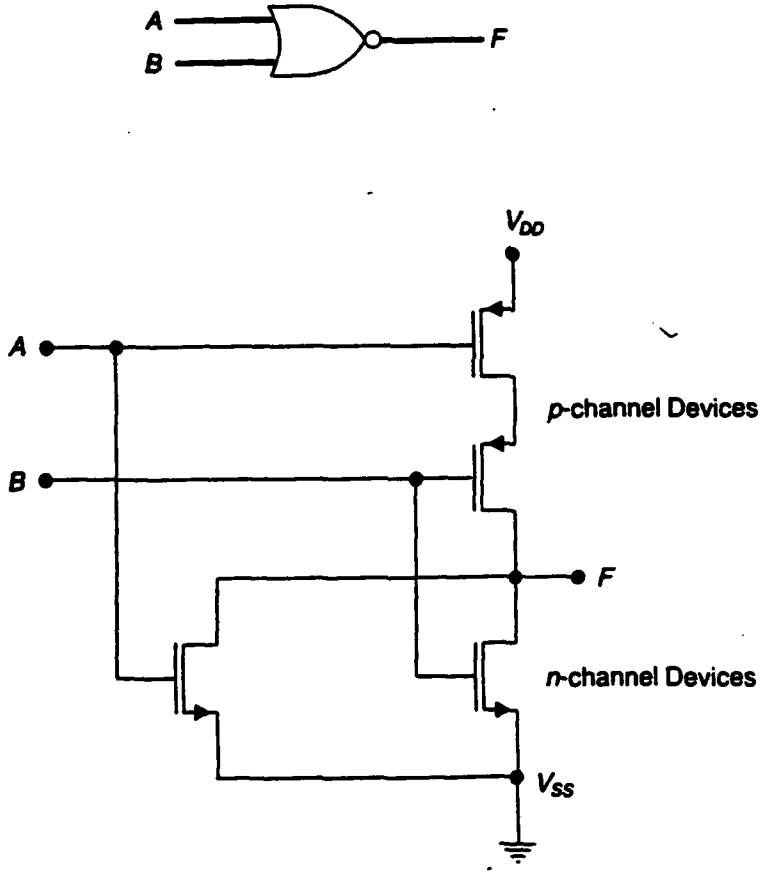


Figure 1.4 Logic diagram and transistor implementation of CMOS NOR gate [Ref. 9]

Several faults, however, do not adhere to the stuck-fault model. One example of such a fault is the **stuck-open fault** [Ref. 1:p. 34]. Consider the NOR gate in Figure 1.5. If a break in a line occurs and the input pattern AB = 10 is presented to the circuit, there is no path from either *VDD* or *VSS* to the circuit output because neither the series *p*-channel transistors nor the parallel *n*-channel transistors is conducting. Consequently, the circuit is floating due to load capacitances, and the output retains its previous value.

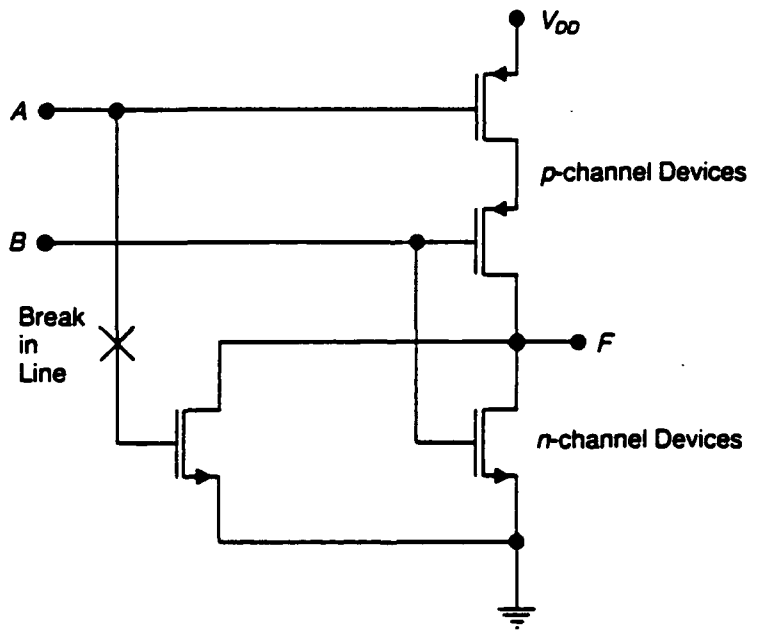


Figure 1.5 Stuck-open fault [Ref. 1]

Because the circuit retains the memory of its previous state, it is no longer a combinational circuit. Instead, the circuit is sequential which violates the second assumption of the stuck-fault model discussed on page nine. As a result, the stuck-open fault cannot be adequately modeled by the stuck-fault model.

There have been attempts to develop new and better fault models to overcome the limitations of the stuck-fault model. For example, logic circuit models of various gates, such as the NOR gate, are described by R.L. Wadsack in 1978 [Ref. 9] to allow the effect of the stuck-open fault to be simulated. However, the complexity of the circuit is substantially increased because for each gate, an additional D-type flip flop and four gates must be added to allow the effects of stuck-open faults to be adequately modeled [Ref. 1:p. 34].

B. GENESIL SILICON COMPILER

The continually growing significance and complexity of VLSI circuits has made it necessary to develop automated design systems to maximize the benefits of this new technology. Design automation provides faster and more efficient methods to design and test integrated circuits. One such state-of-the-art system is the GENESIL Silicon Compiler.

GENESIL is a top-down, hierarchical chip-design method based on silicon compilation, and it is one of the newest Application Specific Integrated Circuit (ASIC) design methods.

Other ASIC design methodologies include full-custom design, gate-array design and standard cell design methods [Ref. 10:p. 38]. For full-custom VLSI design, the circuit designer must have a thorough knowledge of silicon semiconductor technology. However, gate-array design, standard cell design and silicon compilation make VLSI design achievable to systems designers who lack IC designer expertise.

GENESIL is a menu driven interactive layout editing system that concentrates on high-level systems design. There are hundreds of complex functional parts available in its library of cells, such as random access memory (RAM), read only memory (ROM), programmable logic arrays (PLA), arithmetic logic units (ALU), multipliers, basic logic gates and data-path blocks to manipulate parallel data. The designer selects the desired cells and connects them together with the netlist of routing commands.

GENESIL also provides the user with a design verification package which allows the designer to functionally verify the chip design through timing analysis, power requirement analysis and automatic test generation. Hence, the designer is able to quickly and efficiently perform successive design iterations to explore architectural alternatives.

C. THESIS OVERVIEW

The main goal of this thesis is to describe the need for design for testability in a VLSI chip and to provide

information on implementing a DFT strategy to test the multiply-add module of a notch filter using the GENESIL Silicon Compiler. The use of arithmetic codes, specifically a residue code, to check arithmetic operations is the primary concept to be investigated. Chapter II will describe two structured techniques of design for testability: Scan Design methods and Built-in Self Test approaches, including arithmetic codes. Chapter III will describe the basic design of a notch filter and will include a complete functional description of the multiply-add module. This chapter will also describe the methodology used to implement residue code with the GENESIL Silicon Compiler for testing the multiply-add module. Chapter IV will present a summary of the work completed and the conclusions drawn from this research.

II. DESIGN FOR TESTABILITY METHODS

A. BACKGROUND

The testing of sequential devices is quite important due to the frequency of their occurrence in practical designs. In fact, very few complex designs can be achieved using only combinational logic; hence, it is crucial that test techniques for sequential circuits be available.

The basic structure of a sequential circuit is presented in Figure 2.1 for review. This circuit includes both combinational logic and memory elements (usually flip flops). The circuit has sets of *n* primary inputs (X_1, ..., X_n), *m* primary outputs (Z_1, ..., Z_m) and *k* excitation variables (Y_1, ..., Y_k). As can be seen in Figure 2.1, the primary outputs are a function of either the primary inputs or the state variables, or both. The circuit is called a Moore sequential machine if the primary outputs depend only on the state variables [Ref. 1:p. 521]. However, the circuit is called a Mealy sequential machine if the primary outputs depend on both the state variables and the primary inputs [Ref. 1:p. 523]. The next state of the memory elements are specified by the excitation variables which are functions of the primary inputs and the present state variables. Finally, the memory elements use a common clock signal.

Sequential circuits require verification that the circuit provides the correct primary outputs for a given set of primary inputs, and the circuit also requires verification that the correct state transition occurs. For this reason, Sequential circuits are extremely difficult to test.

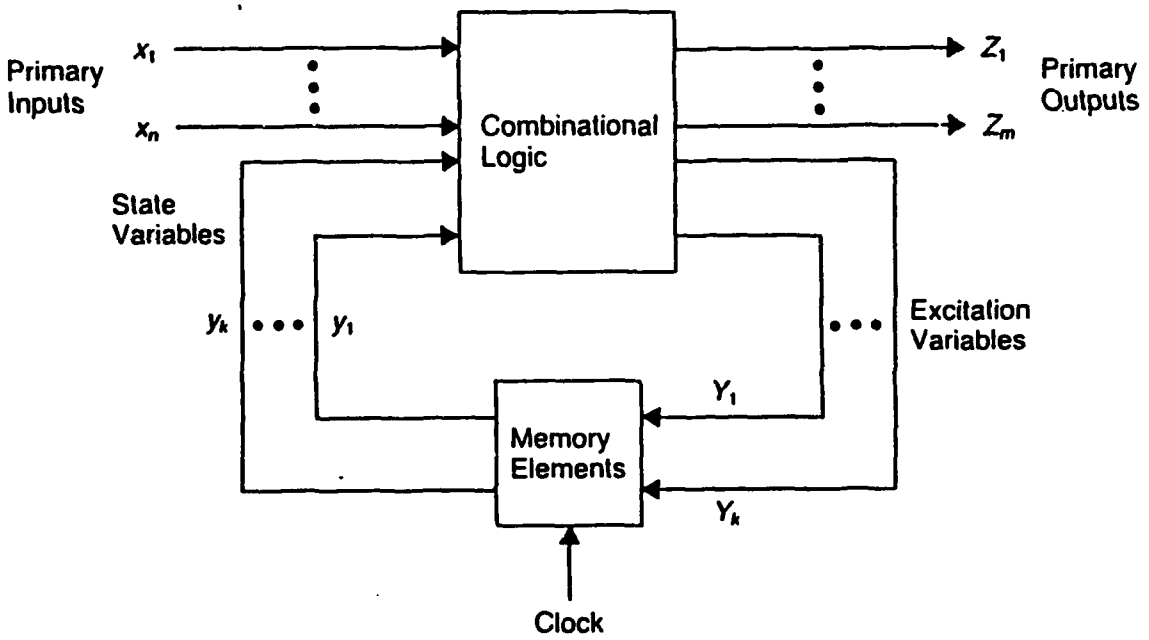


Figure 2.1 Basic structure of a sequential circuit [Ref. 1]

As k (excitation variables) and n (primary inputs) become large, the testing of sequential machines becomes prohibitively complex because the primary outputs and the state transitions for all possible primary inputs and all possible initial states must be verified to completely test a sequential machine [Ref. 1:p. 523]. In fact, it is impossible from a practical standpoint to completely, functionally test today's sequential devices unless some constraints are placed on their design. Constraints on the design of digital circuits are one form of design for testability [Ref. 3]. The objective of DFT is to create a design that is easy and economical to test.

DFT techniques can be divided into ad hoc methods and structured approaches [Ref. 1:p. 523]. The ad hoc techniques include heuristic methods such as circuit partitioning and adding extra test points. In the circuit partitioning method, the circuit is separated in several small, independently tested modules. The idea behind this approach is that testing several small circuits is much easier than testing one large, complicated circuit. Test points are used to improve the testability of a circuit by allowing an external test device to have easy access to internal nodes of the circuit for control and observation. Ad hoc approaches can be adequate for a specific design, but they are generally not applicable to all designs. Furthermore, there is very little standardization when using ad hoc methods.

Structured DFT techniques, on the other hand, involve a set of general design rules by which a design is implemented. The same set of design rules are typically required for all designs. Thus, structured design for testability improves the ability to test sequential machines, and it standardizes designs [Ref. 1:p. 523].

Most structured DFT techniques convert sequential machines into combinational circuits for testing by providing a means of breaking the feedback loop in the sequential machine. This allows the state of the machine to be controlled and observed for easy verification of correct operation. Figure 2.2 shows this concept of controlling the state variables of the circuit and observing the excitation variables. The test process is then simplified to one of testing the combinational logic which has inputs consisting of the state variables and the primary inputs and which has outputs consisting of the excitation variables and the primary outputs. Hence, all inputs to the combinational logic become completely controllable and all outputs become completely observable with the feedback loop broken and the state variables accessible [Ref. 1:p. 524].

In this chapter, two structured design for testability techniques will be examined: scan path and built-in self test. These two techniques can be used separately, or both can be included in a single VLSI design, and these techniques are implemented on GENESIL by using a **Testability Latch Block**.

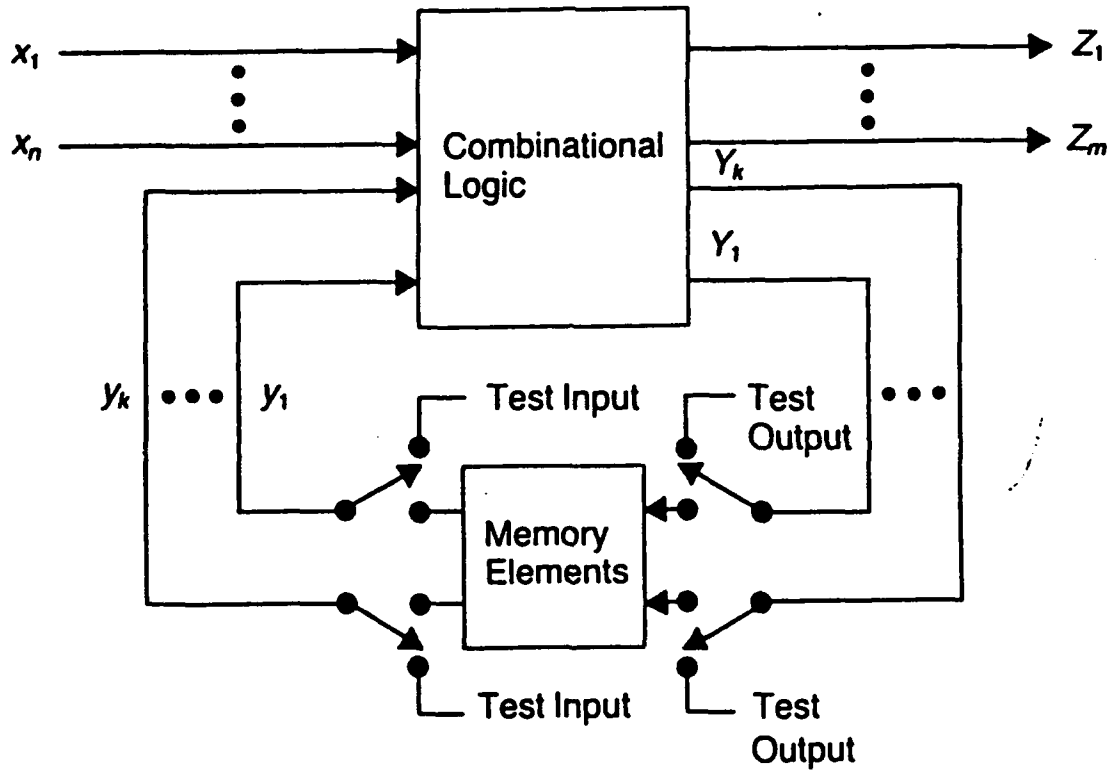


Figure 2.2 Structured DFT concept of converting a sequential machine into a combinational circuit [Ref. 1]

For parallel datapath designs, three configurations of the Testability Latch Block are available in GENESIL:

1. The basic configuration which uses a single shift register to serially enter or retrieve data.
2. The generator configuration which has the attributes of the basic configuration as well as including circuitry for pseudorandom test sequence generation.
3. The signature configuration which has the attributes of the generator configuration plus signature analysis logic circuitry.

The first configuration is used to implement the scan design techniques while the latter two configurations implement the built-in self test techniques through use of linear feedback shift registers [Ref. 11:p. 24-2].

B. SCAN DESIGN

The object of design for testability is to find ways to make controllability and observability of the object under test easier, more efficient and less costly. **Scan design** is the approach used in methods such as Scan Path, Scan Set and Level Sensitive Scan Design (LSSD), and it requires the use of specially designed, clocked flip flops that can be placed in either the *operate* or *test* mode [Ref. 1:p. 524]. These flip

flops are able to accept test vectors that control the present state of the circuit, and they are able to clock (or scan) out the current excitation variables of the circuit. By interconnecting all the flip flops into a single shift register, scanning techniques can clock an appropriate test vector into this shift register, perform a normal operation (usually one clock cycle) and shift out the resulting excitation variables of the circuit [Ref. 1:p. 525].

A generalized block diagram of a system using a scanning method of design for testability is shown in Figure 2.3. The individual flip flops of the shift register operate independently and perform normal system functions during normal operation, but each flip flop can be loaded with a specific value by shifting in a serial data stream through the scan-in line during the test process [Ref. 1:p. 525].

After the flip flops are loaded for a test process, the combinational circuit can perform a normal operation. The (single cycle) primary outputs can then be observed in a normal fashion, and the excitation variables can be observed by loading their values into the shift register and shifting out the result via the scan-out line [Ref. 1:p. 525].

Scan design enhances the controllability problem through its ability to shift in data to internal nodes and, scan design enhances the observability problem through its ability to access test results via the scan-out line. Furthermore, this accessibility to internal nodes of the circuit is gained

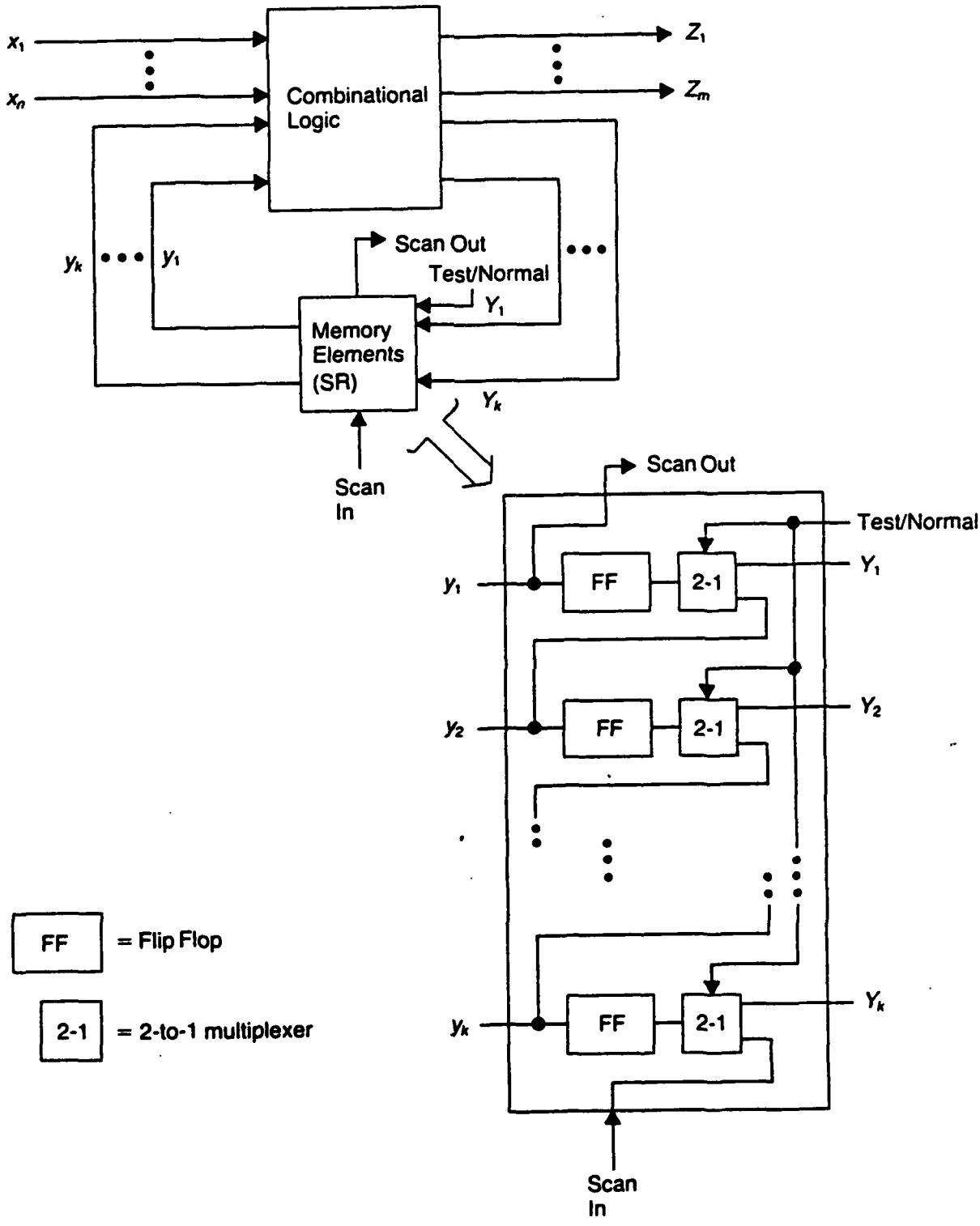


Figure 2.3 Block diagram of system using scan design [Ref. 1]

using a minimal number of peripheral pins for testing because only the serial input of the first shift register and the serial output of the last register are required for vector manipulation.

Scan Path is a method of design for testability developed by Nippon Electric Co., Ltd. [Ref. 12]. This approach partitions the design of a large circuit into subsystems of Scan Path registers connected together. Each subsystem can be uniquely enabled for test purposes, thereby, allowing portions of a system to be independently tested to a higher degree and in an easier manner than the design as a whole. Figure 2.4 shows how a generalized circuit might be partitioned by scan path into individually testable subsystems [Ref. 13:p. 374].

Internal flip flops in the data path are replaced with master-slave configured D-type flip flops that can be chained together to make the needed shift registers [Ref. 1:p. 529]. The design uses a single clock signal that controls two latches. The clock signal for the first latch goes through an inverter to become the clock signal for the second latch. A disadvantage to this approach is that race conditions could occur if the input to the D-type flip flop changes at approximately the same time that the clock changes or if the output of the second latch feeds back through combinational logic to become the input to the first latch [Ref. 3:p. 105]. However, the problems mentioned above can be avoided by adhering to appropriate design rules.

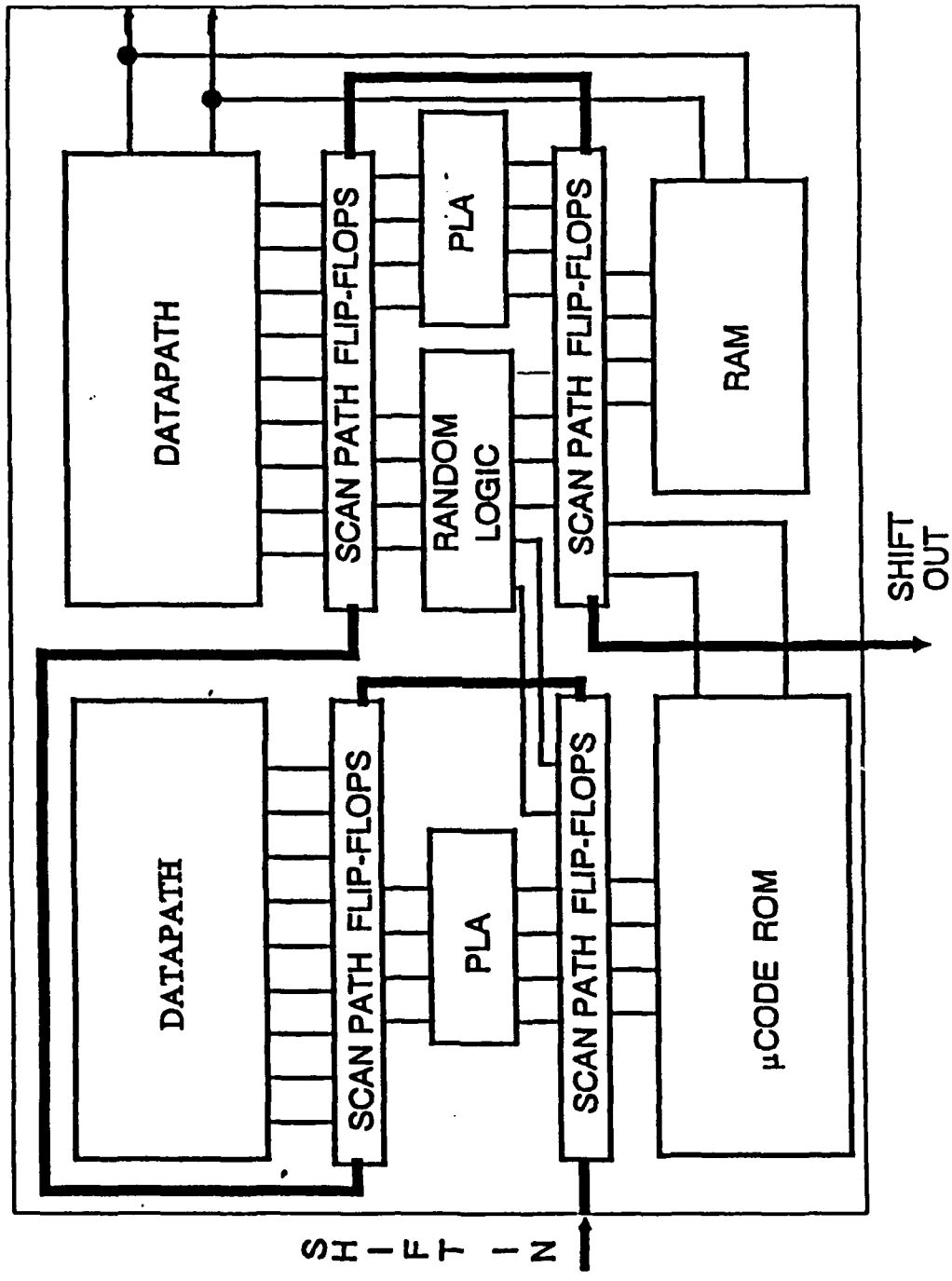


Figure 2.4 General circuit partitioned by scan path [Ref. 13]

Level Sensitive Scan Design (LSSD) is a technique developed by IBM to overcome potential race problems. The term *level sensitive* means that the steady-state response to an allowable input change is independent of delays within the circuit and the order which signals change within the circuit. The term *scan design* implies that the technique uses the scanning approach [Ref. 1:p. 525].

LSSD adheres to the same basic idea as Scan Path for moving test vectors into and out of the circuit, but the structure of the flip flops used to construct the shift register is fundamentally different. The master-slave D-type flip flop shown in Figure 2.5 is the key element used in the LSSD methodology. The flip flop is a master-slave D-type flip flop which uses two non-overlapping clocks and is provided with an extra input stage, allowing the input to the master flip flop to come from either the normal line or test line. The test input stage of the flip flop is disabled during normal operation and the circuit performs as a normal master-slave D-type flip flop. The normal input stage can be disabled during testing to allow the flip flop to be loaded with the test input [Ref. 1:p. 525].

The LSSD technique overcomes the potential race problem present in the Scan Path technique by using separate, non-overlapping clocks for the master and slave flip flops to provide the level-sensitive operation [Ref. 3:p. 105]. A third clock input is provided for the scan operation which is

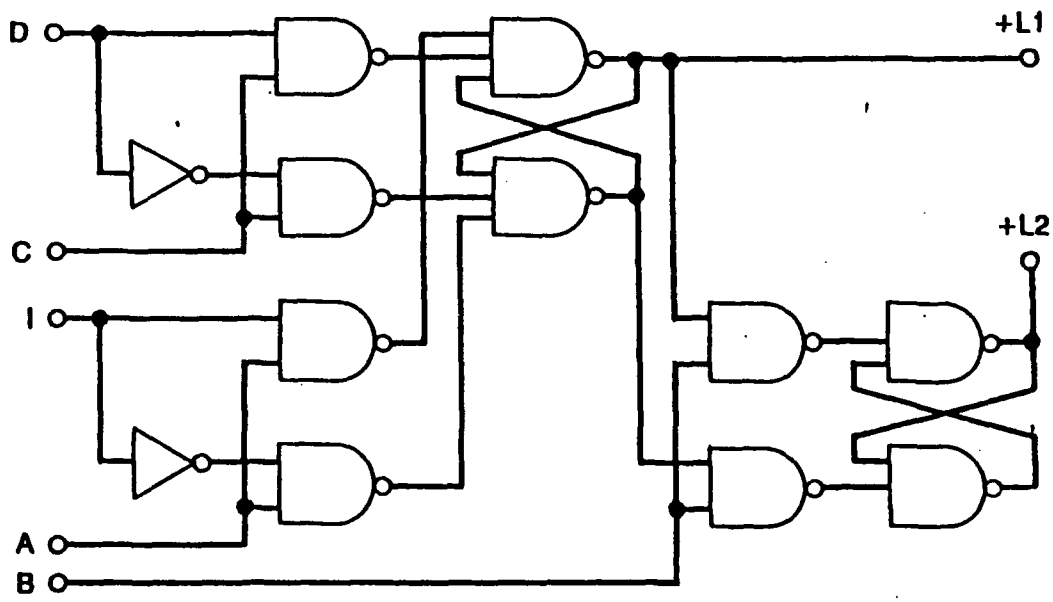


Figure 2.5 LSSD master-slave D-type flip flop [Ref. 1]

also accomplished using non-overlapping, two-phase clocking [Ref. 1:p. 526]. The flip flops are chained together throughout the system by connecting the outputs of the slave portion of the flip flop to the scan input of the master section of the next flip flop to provide the scan feature shown in Figure 2.6. During normal operation, clocks C and B of Figure 2.5 are used in the master (L1) and slave (L2), respectively. Whereas, during the testing or scan operation, clocks A and B are used. The test input is line I.

Scan Set is a technique similar to Scan Path and LSSD developed by Sperry-Univac to enhance the testability of a design [Ref. 14]. However, the shift register in Scan Set is not located in the data path. Instead, the shift register in Scan Set is an additional component that is completely independent of the normal system flip flops [Ref. 1:p. 532].

As shown in Figure 2.7, the shift register in Scan Set can be used to sample the values of various points within the circuit and control the values of certain lines. A test vector can be clocked into the shift register via the scan-in line and then applied to the circuits' logic, and values of the circuits' response on certain lines can be sampled by the shift register and clocked out on the scan-out line. The flip flops used to provide the normal system operation must be multiplexed with the test inputs provided by the shift register [Ref. 1:p. 533].

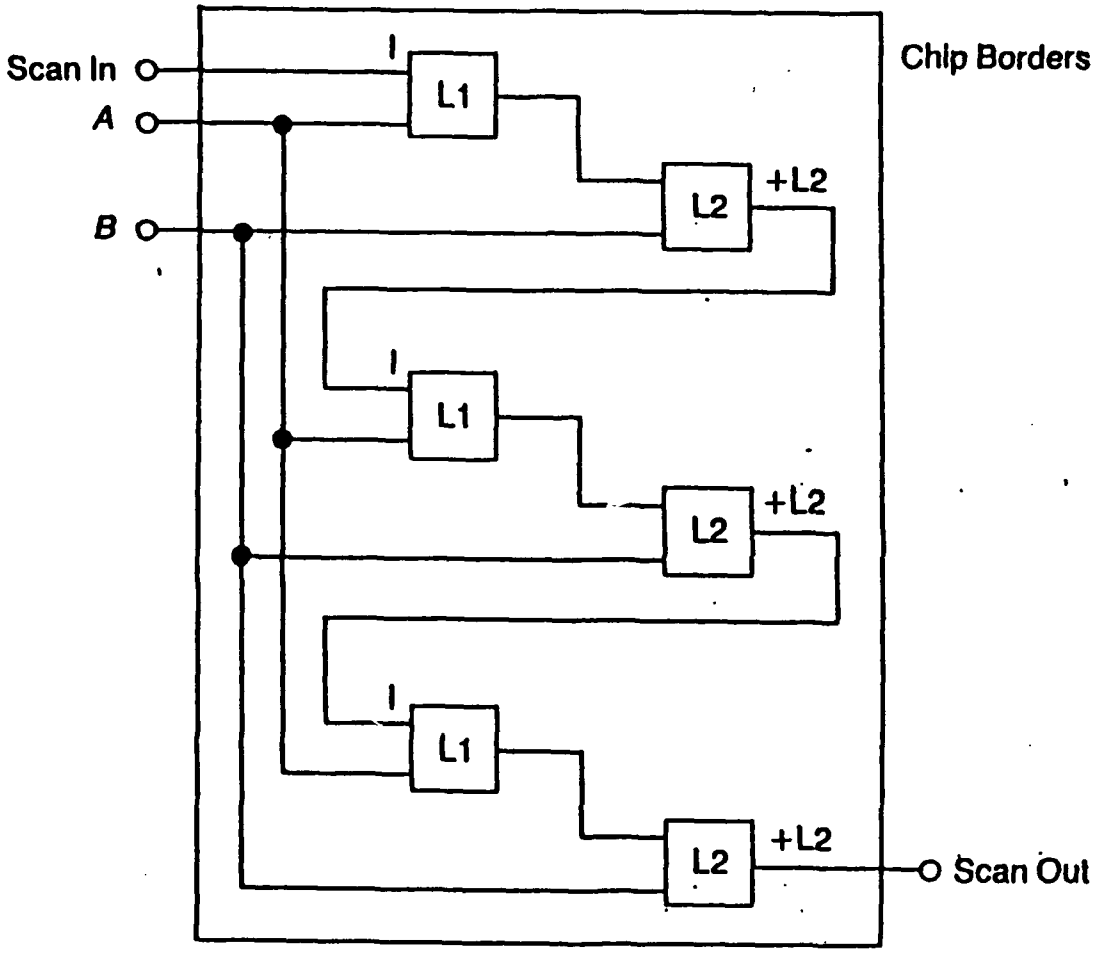


Figure 2.6 Formation of shift register in LSSD [Ref.1]

There are two advantages to the Scan Set method. First, a designer can determine exactly which of the sequential circuit latches he desires to have the ability to set if he does not desire the ability to set all latch points. Second, points within the circuit can be observed during normal operation because the scan shift register is not an integral part of the system [Ref. 3:p. 106].

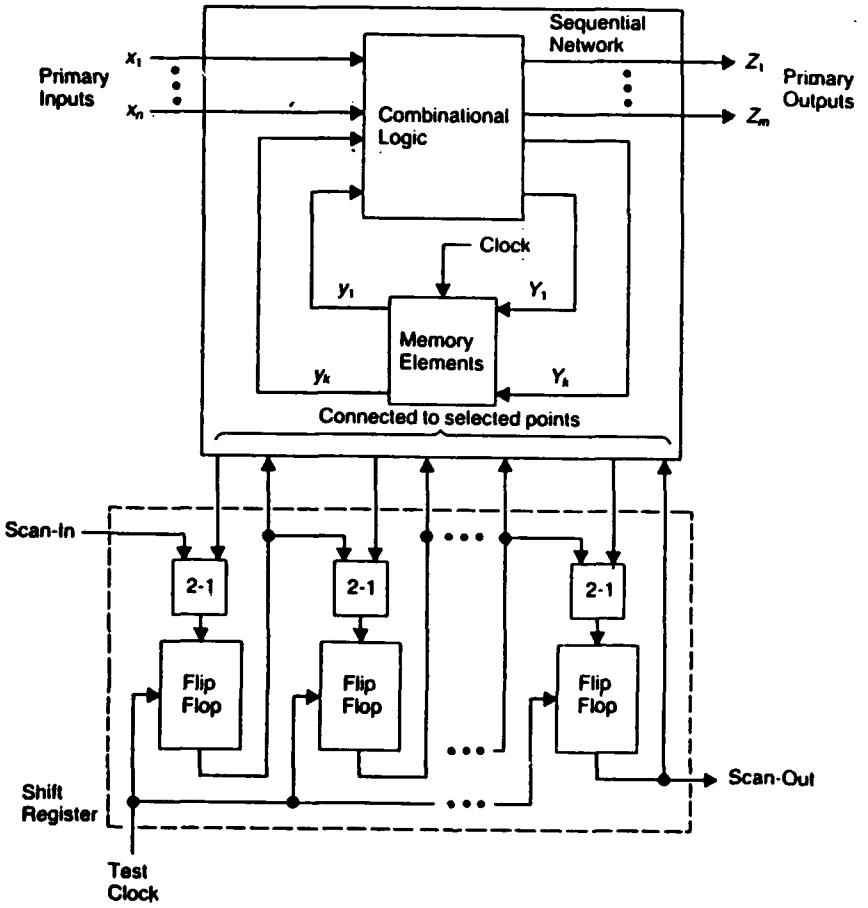


Figure 2.7 Scan Set logic [Ref. 1]

C. BUILT-IN SELF TEST DESIGN

The second structured design for testabilit technique to be examined is built-in self test or built-in test. External test techniques, such as scan design, require that the circuit under test be removed from its operational environment and tested by external equipment. With ever increasing circuit densities due to VLSI, the number of test patterns required for exercising a circuit under test is becoming too large to be efficiently handled by external test equipment. Also, the time required to generate and apply the test patterns is growing too large [Ref. 15:p. 21].

Built-in self test (BIT) techniques attempt to overcome the problems of external testing by incorporating some or all of the tester functions into the design of the device such that testing can be accomplished without external test equipment. The scan-in-and-out of data for each single-cycle test can be avoided by using *internal pattern generation* (source) and *response compaction* (sink). BIT techniques typically use a *pseudorandom-pattern generator* to generate test vector patterns and a *pattern compactor* to compact response patterns. The pseudorandom-pattern generator and the pattern compactor are both located within the circuit to be tested, or in close proximity to it, and they permit built-in test techniques to achieve in-system at-speed testability [Ref. 2:p. 170].

The **Linear Feedback Shift Register** (LFSR) is the most common device used for generating pseudorandom test patterns for BIT techniques. The test patterns are called "pseudo"-random because they are generated in a predetermined *sequential order*, depending on initialization values and actual implementation of the LFSR [Ref. 2:p. 172].

Figure 2.8 shows the basic structure of a linear feedback shift register implemented by GENESIL. It consists of an n-stage (n-bit-position) shift register where outputs of the last and some *intermediate* stages are fed back through XOR gates to the *first* stage. The R input is the multiplexer control line, and it determines whether data is shifted into the least significant bit (bit 0 stage) from the serial input line (TIN) or from the XOR feedback path. The linear feedback shift register is initialized by serially loading the desired value into each stage, or bit position. After the initialization is completed, the multiplexer control line is selected so that data is shifted into the least significant bit from the XOR feedback path only. As a result, the LFSR stages assume different contents with each shift-cycle, starting with the initial content, and will generate a pseudorandom sequence of cyclic periodicity [Ref. 2:p. 560]. This sequence can then be used as internally generated test patterns for a circuit under test.

The values of a_i, such as $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$, in the LFSR polynomial determine how the 2-input-XOR gates are

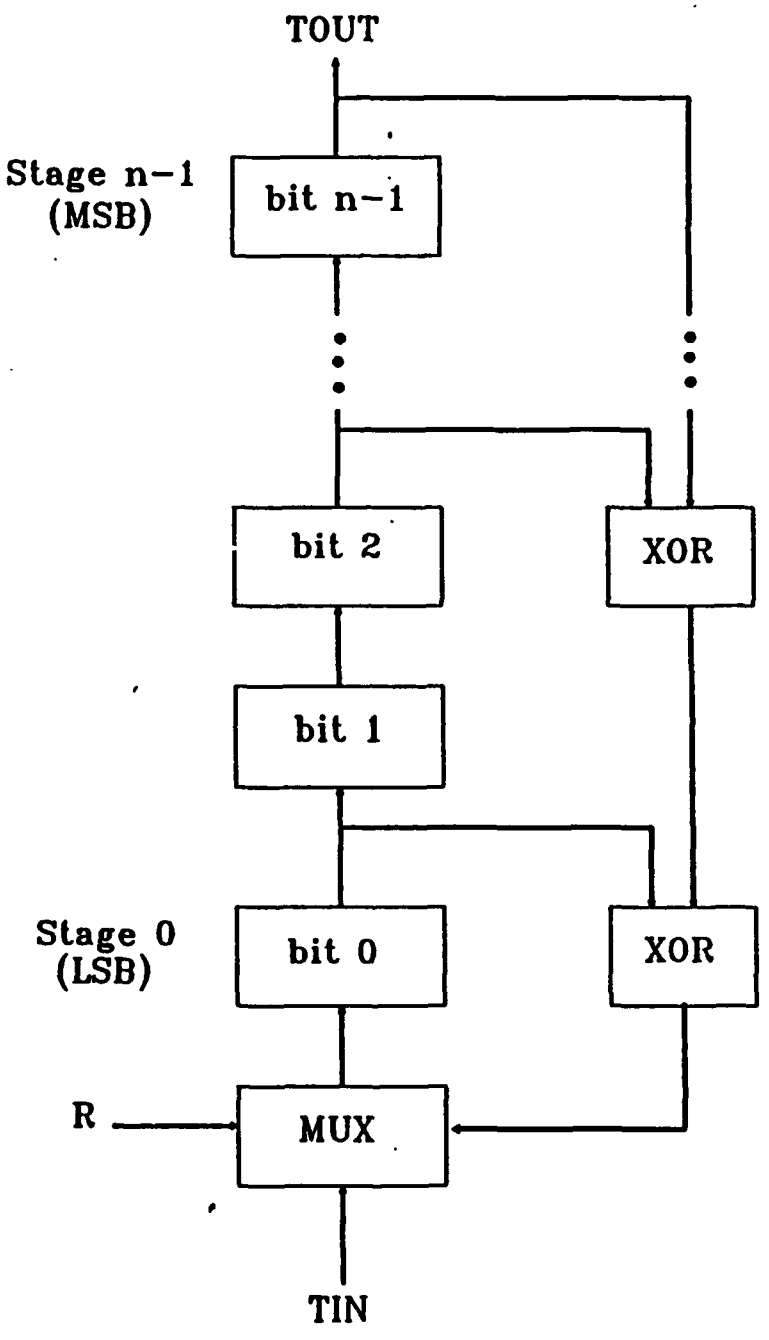


Figure 2.8 Basic structure of LFSR [Ref. 7]

incorporated in the feedback paths, and these values are fixed for a particular implementation. This polynomial also determines the cyclic periodicity or length of the pseudorandom test sequence. A polynomial of degree n with a *minimum* number of non-zero coefficients is an "irreducible, primitive" polynomial and represents the most economical realization for an n-stage LFSR because a minimum number of 2-input-XOR gates is incorporated in the feedback paths [Ref. 2:p. 174]. Furthermore, "irreducible, primitive" polynomials can be used for designing "*maximum-length*" sequence generators. As a general rule, the maximum-length sequence generator for a n-stage LFSR can generate $[2^n - 1]$ unique n-bit-long test patterns [Ref. 2:p. 175].

The execution of a long sequence of tests in rapid succession is made possible by compressing the test response patterns; otherwise, a large storage capacity would be required for collecting the successive test results [Ref. 2:p. 177]. The compressing, or encoding, of a large amount of digital information into a fixed small *signature* that characterizes the response of the circuit under test is the basic concept of **signature analysis** [Ref. 1:p. 517].

The LFSR can be the encoding circuit used for collecting and compressing test response patterns in the signature analysis approach, and for that reason the LFSR is called a signature register. As shown in Figure 2.9, the signature register is implemented by inserting an XOR gate in the

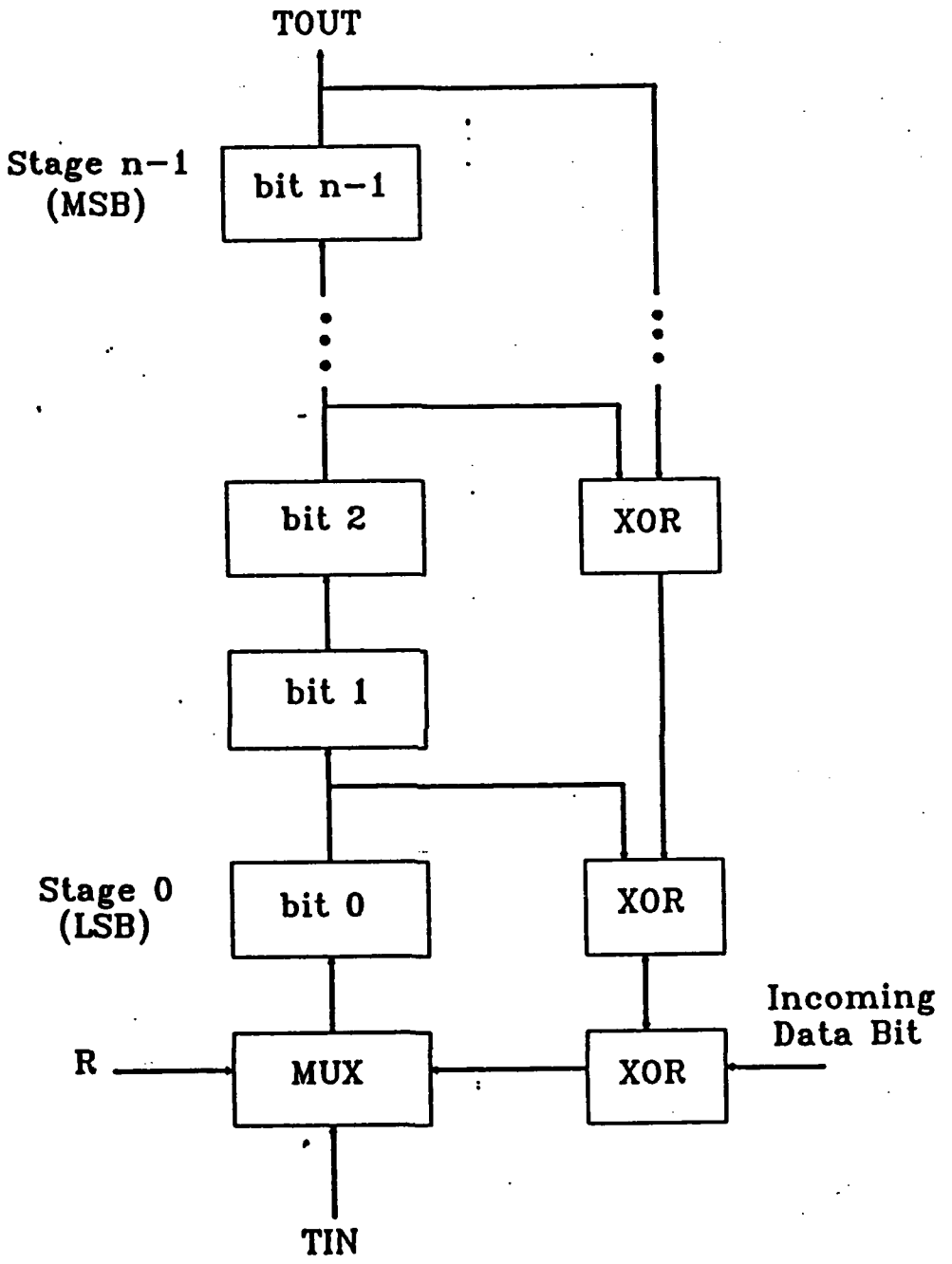


Figure 2.9 Signature register [Ref. 8]

feedback path prior to the input to the first stage to allow modulo-2 addition of the incoming data stream and the feedback path from the other XOR gates [Ref. 2:p. 564]. The content of the signature register is usually called the residue or the syndrome, and it is determined by the content of the register prior to the occurrence of a clock pulse and the value of the serial data input line. Therefore, the final content, or syndrome, of the signature register is determined by the input bit pattern. The resulting syndrome is the signature used in signature analysis [Ref. 1:p 519]. If a fault occurs, the output bit sequence will change, resulting in a different signature in the signature register.

In signature analysis, signature registers are placed at specific points within the circuit under test such as shown in Figure 2.10. A given input test sequence is then applied to record the signature of the circuit under test. These signatures are compared to known *good* signatures of a fault-free circuit. If the signatures of the good circuit disagree with those of the circuit under test, the circuit under test is considered faulty [Ref. 1:p. 518].

Arithmetic codes are fault detection techniques that can be used to provide concurrent, built-in test. An arithmetic code is a redundant representation of numbers, and certain errors in arithmetic operations can be detected using these numbers [Ref. 16:p. 65]. To accomplish this fault detection, the data is encoded before the arithmetic operations are

performed, and the code words resulting from the arithmetic operations are checked for validity [Ref 1:p. 112]. If the code words are not valid, an error condition exists.

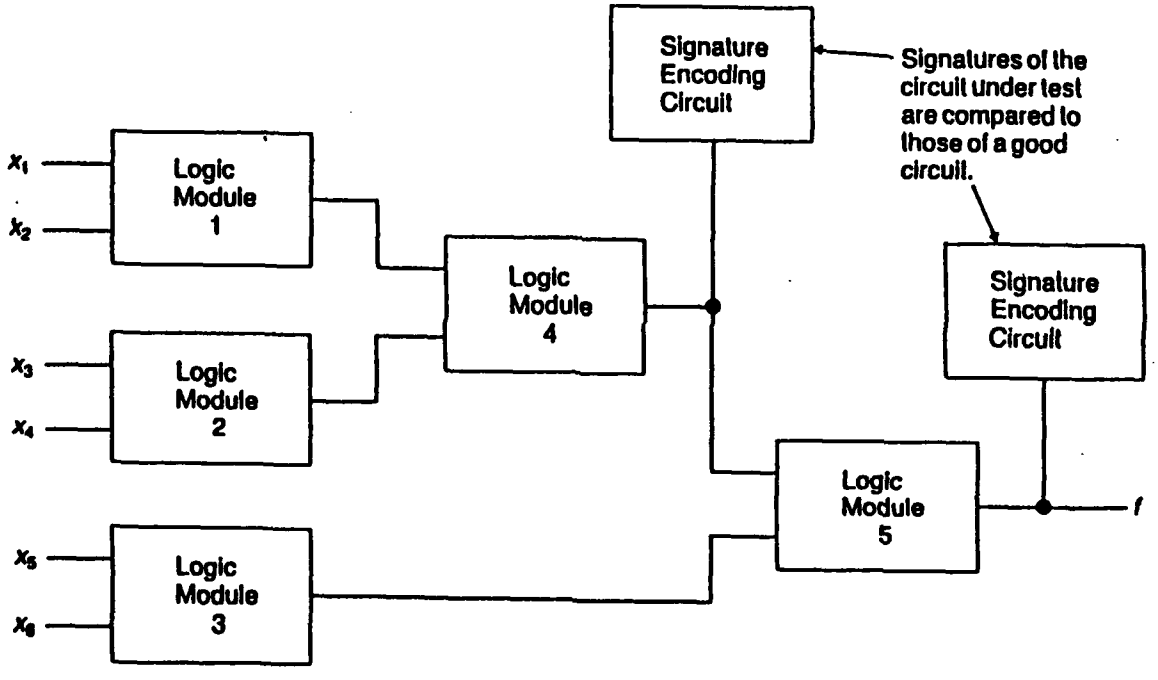


Figure 2.10 Placement of signature registers [Ref. 1]

An arithmetic code is preserved during the arithmetic operation [Ref. 17]. Given two numbers *b* and *c* and an arithmetic operation *, then A is an arithmetic code with respect to the operation * if A(*b* * *c*) = A(*b*) * A(*c*), where A(*b*) and A(*c*) are arithmetic code words for the numbers *b* and *c*, respectively [Ref. 1:p. 112]. In other words, an arithmetic operation performed on two arithmetic code words will produce the arithmetic code word of the arithmetic operation.

Arithmetic codes provide at-speed testing for detection of transient and permanent faults concurrent with system operation [Ref. 18:p. 325]. However, the economic feasibility of an arithmetic code is determined by the cost and effectiveness of the arithmetic operations provided and the speed requirement. Examples of arithmetic codes are AN codes, residue codes, inverse residue codes and the residue number system. Residue codes will be discussed in further detail because this code was the approach chosen for this thesis to incorporate a design for testability strategy into the multiply-add module of a notch filter.

A residue code is a *separable* arithmetic code. In a separable code, the check bits are separated from the number (operand). The code word is usually generated by appending the residue of a number to that number. For example, a code word can be represented as D/R, where D is the data and R is the residue of that data [Ref. 1:p. 115].

The residue of a number is the remainder produced when that number is divided by an integer called the check base, or the modulus. For example, if the original number is 10 and the modulus is 3, the quotient will be 3 and the residue will be 1. This example is often written as:

$$10 = 1 \text{ modulo } (3)$$

This is stated as ten is congruent to one modulo three. The number of extra bits that are appended to a data word to represent a separable residue code word depends on the modulus, but the residue will never be larger than the modulus [Ref. 1:p. 116]. Table 2.1 shows the residue code produced when 4-bit information words are encoded using a modulus of 3.

Residue codes are very useful for checking arithmetic operations because the residues can be handled separately from the data. Figure 2.11 shows how a separable residue code can be implemented to provide error detection for an adder. D_1 and D_2 are two data words added to form a sum word s. r_1 and r_2 are the residues of D_1 and D_2, respectively, and these residues are added using a modulo-m adder. Modulus m is also used to encode D_1 and D_2. If there are no errors, the modulo-m addition of r_1 and r_2 yields r_s which should equal r_c, the residue of the sum s. However, if r_s differs from r_c an error has occurred [Ref. 1:p. 117].

Table 2.1 Residue code words for 4-bit data words using a modulus of three [Ref. 1]

Information	Residue	Code word
0000	0	0000 00
0001	1	0001 01
0010	2	0010 10
0011	0	0011 00
0100	1	0100 01
0101	2	0101 10
0110	0	0110 00
0111	1	0111 01
1000	2	1000 10
1001	0	1001 00
1010	1	1010 01
1011	2	1011 10
1100	0	1100 00
1101	1	1101 01
1110	2	1110 10
1111	0	1111 00

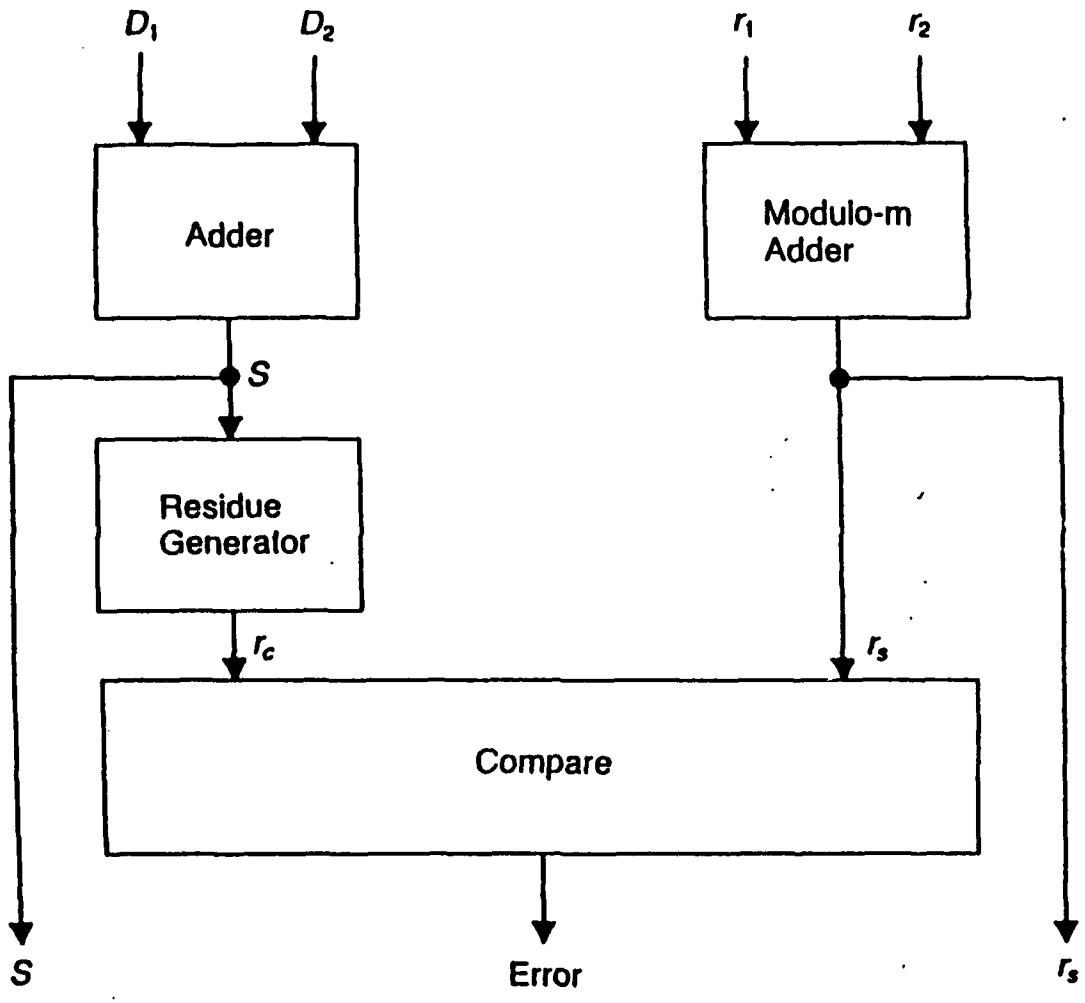


Figure 2.11 Error detection using residue code [Ref. 1]

As defined by Avizienis [Ref. 17], low-cost residue codes have a modulus of $m = 2^b - 1$, where b is some integer greater than or equal to 2 and is called the *group length* of the code. The number of extra bits appended to a data word to represent a code word in a low-cost residue code is equal to b. A low-cost residue code makes the encoding process easy because the division required to find the residue is recast as an addition process due to the congruence

$$Kr^i = K \text{ modulo } (r - 1)$$

Where $r = 2^b$, since $r = 1$ modulo $(r - 1)$. Accordingly, the residue for a kb-bit data word can be obtained by adding the kb-bit groups with an addition algorithm which "casts out $2^b - 1$'s" [Ref. 18:p. 335]. For example, the data bits that are to be encoded in Figure 2.12 are divided into groups containing b bits, and then the groups are successively added in a modulo-$(2^b - 1)$ fashion to form the residue [Ref. 1:p. 118]. Figure 2.13 shows how the residue for an eight-bit data word can be generated using three, 2-bit, modulo-3 adders.

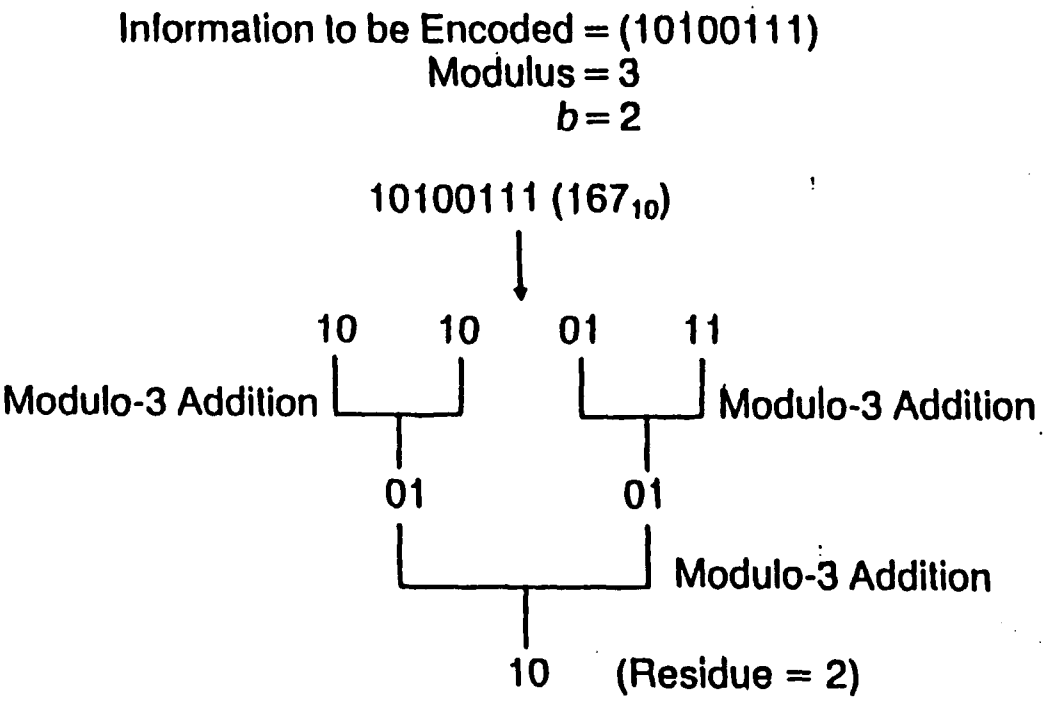


Figure 2.12 Low-cost residue code calculation [Ref. 1]

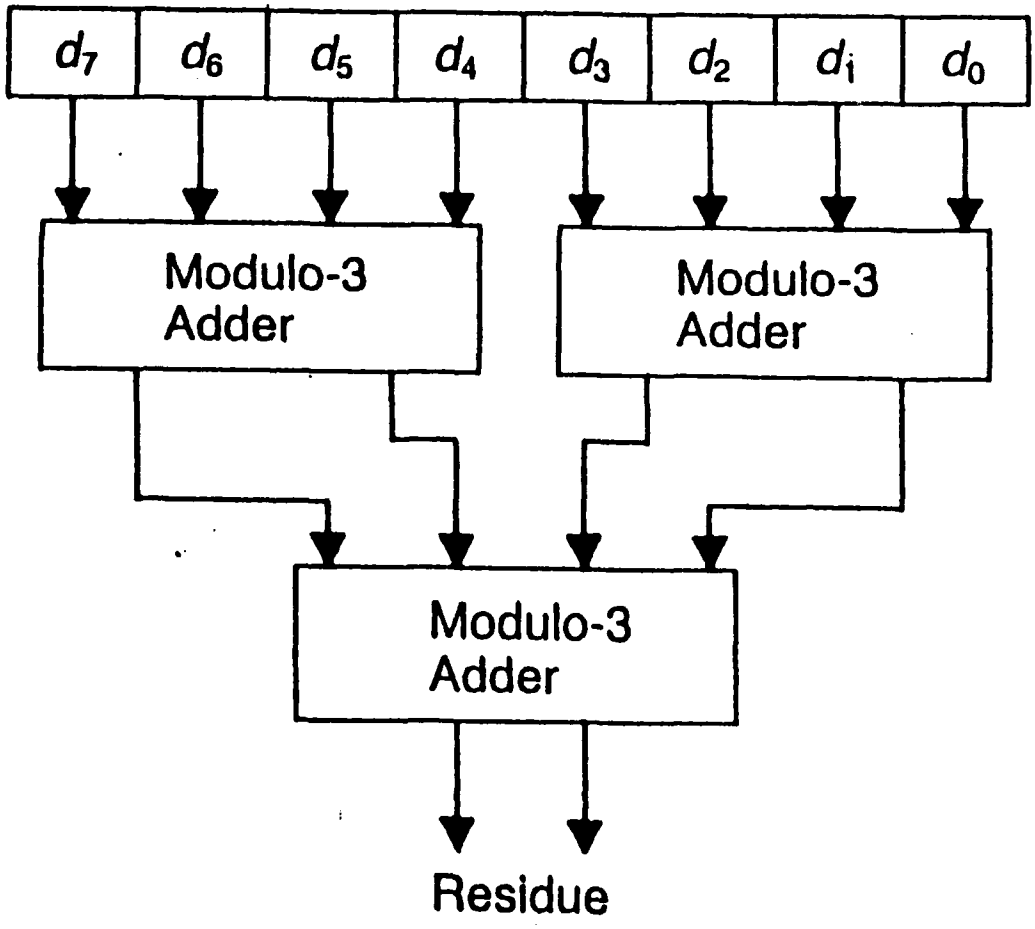


Figure 2.13 Residue generation for an 8 bit word [Ref. 1]

III. IMPLEMENTATION OF A DESIGN FOR TESTABILITY STRATEGY

A. FUNCTIONAL DESCRIPTION

The VLSI chip chosen for implementation of a design for testability strategy was the multiply-add module of a notch filter designed by LCDR Chih-fu Kung, Taiwan Navy, at the Naval Postgraduate School in Monterey, California [Ref. 19]. Figure 3.1 shows a basic second-order Infinite Impulse Response (IIR) notch filter, and from this block diagram it can be seen that the multiply-add module is the fundamental building block for the notch filter.

A basic multiply-add module is shown in Figure 3.2. The multiplier is represented by a triangle while the adder is represented by a circle. The multiply-add module multiplies a fixed-input (x[t]) by a constant(a) and adds the result to another input stream (y[t]), thus producing the output stream (z[t]). This operation can be represented as z[t] = a * x[t] + y[t].

There are four number systems that can be used to represent negative numbers: signed magnitude, ones' complement, two's complement and excess 2^m - 1 [Ref. 19:p. 13]. The ones' complement number system is used for this thesis.

A ones' complement, multiply-add module consists of six blocks: coefficient block, ones' complement to signed

magnitude block, multiplier, signed magnitude to ones' complement block, adder and overflow block. Figure 3.3 is a block diagram of these sections.

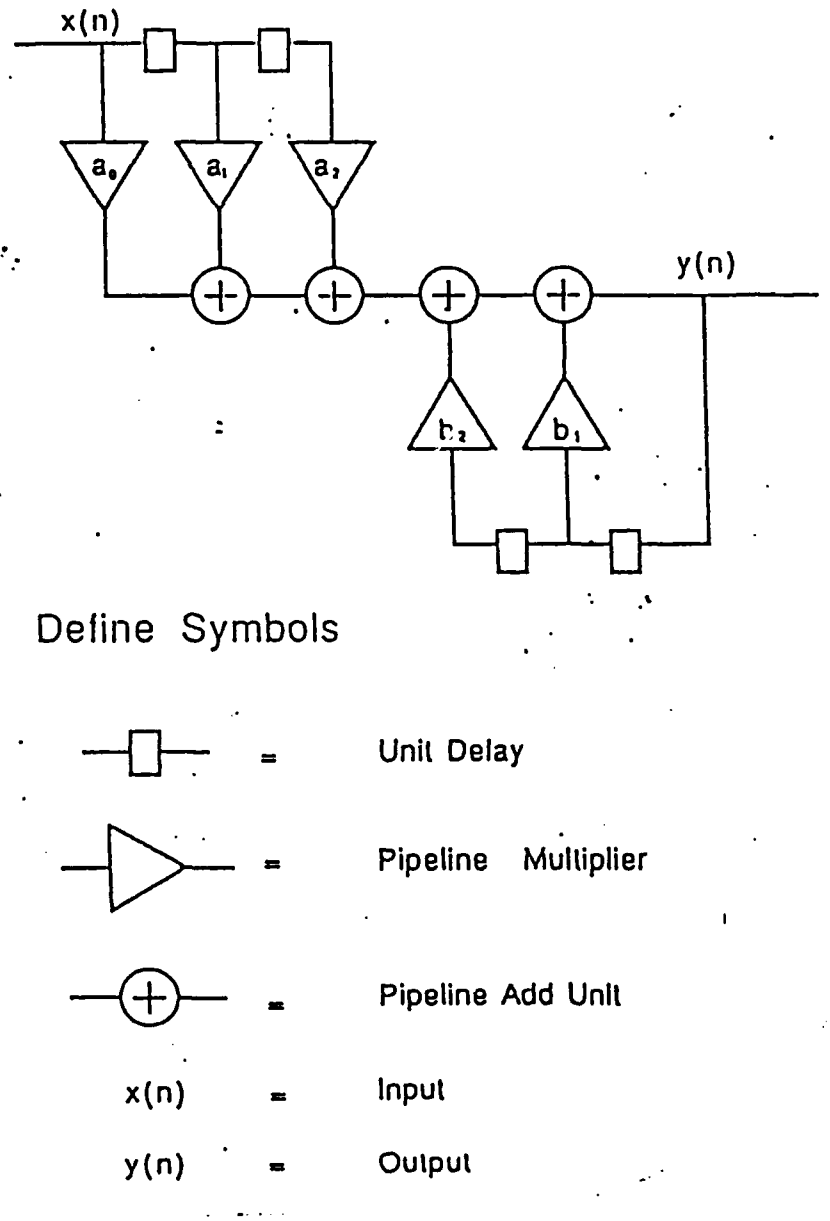
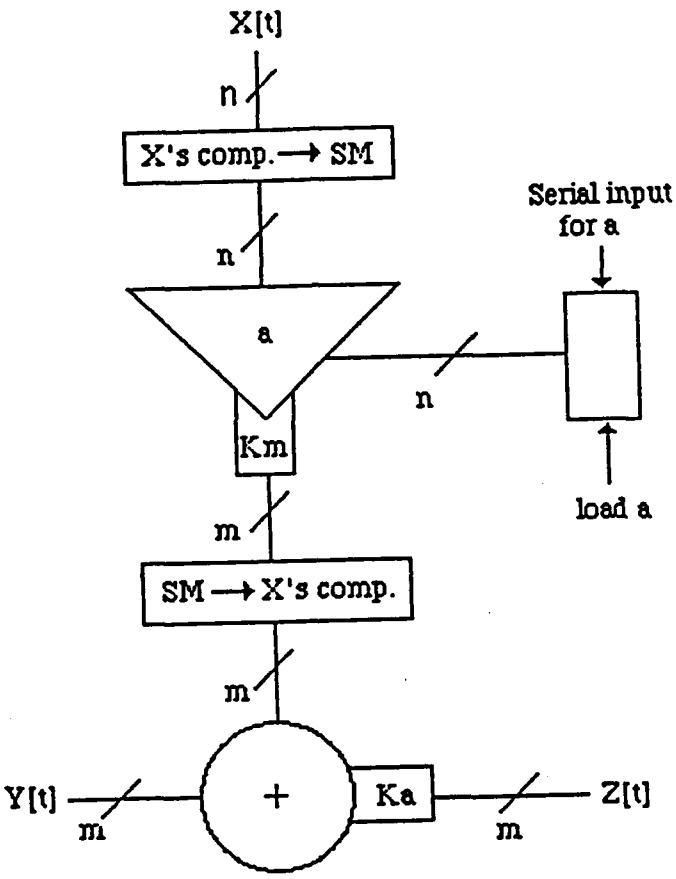


Figure 3.1 Second-order IIR notch filter [Ref. 19]



Definitions

$a = a_s a_0 . a_1 \ldots\ldots a_{n-2}$

$X[t] = X_s . X_1 X_2 \ldots\ldots X_{n-1}$

$Z[t] = z_s z_{-1} z_0 . z_1 z_2 \ldots\ldots z_{n-3}$

Figure 3.2 Basic multiply-add module [Ref. 19]

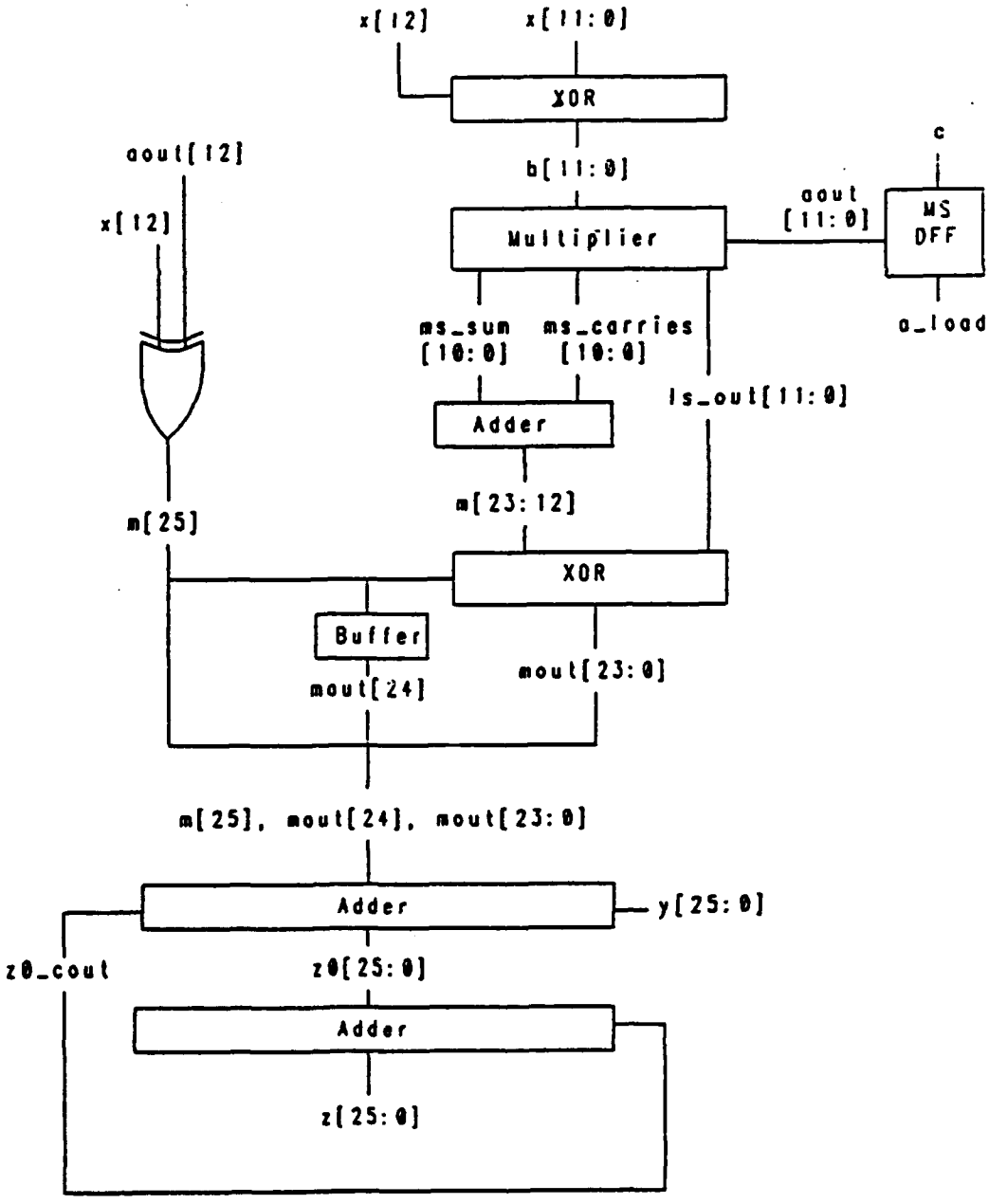


Figure 3.3 Ones' complement multiply-add module

1. Coefficient Block

The constant coefficient (a) is serially loaded to reduce the I/O pin requirement. This block is designed as a serial-in/parallel-out register which requires only two pins for loading one coefficient: one pin for information and one pin for control. Figure 3.4 shows the diagram of a four-bit serial-in/parallel-out register for a coefficient block. For the notch filter design, a 13-bit constant coefficient (a) is used, and this constant could be either positive or negative. The fixed-point signed magnitude format of the constant is represented as $a = a_s\ a_0\ a_1\ a_2\ \ldots a_{11}$, where a_s is the sign bit.

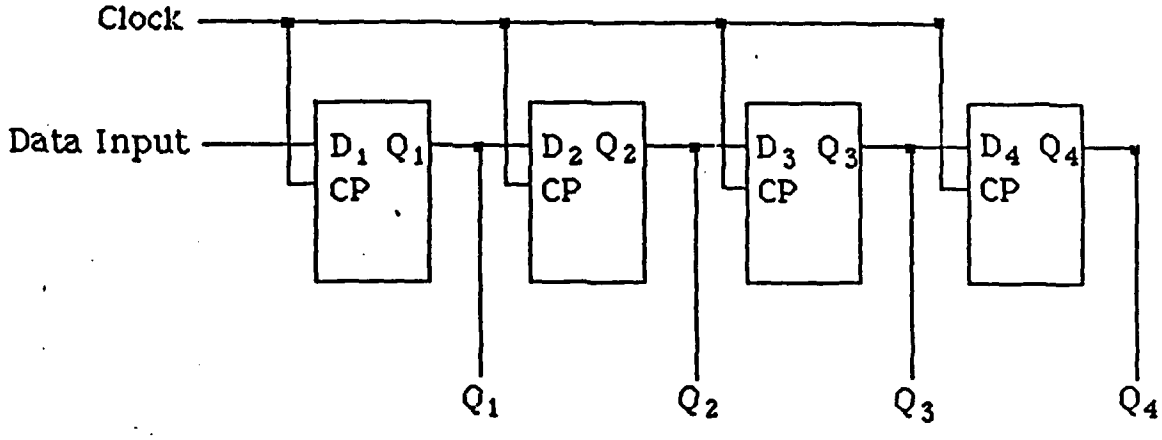


Figure 3.4 Four-bit serial-in/parallel-out register [Ref. 19]

2. Ones' Complement to Signed Magnitude Block

The sign bit is positioned as the leftmost bit. Positive numbers are represented the same for ones' complement and signed magnitude; however, for negative numbers in ones' complement the sign is one and the remaining bits are the complement of the magnitude. As a result, the conversion from ones' complement to signed magnitude is simple: do nothing for positive numbers and take the bit-by-bit complement for negative numbers [Ref. 19:p. 14]. By checking the sign bit and selectively complementing the magnitude through XOR gates, the conversion from ones' complement to signed magnitude can easily be achieved because a two-input XOR gate has no effect when one of its inputs is zero (a positive number) but acts as an inverter when one of its inputs is one (a negative number). This conversion process is illustrated in Figure 3.5.

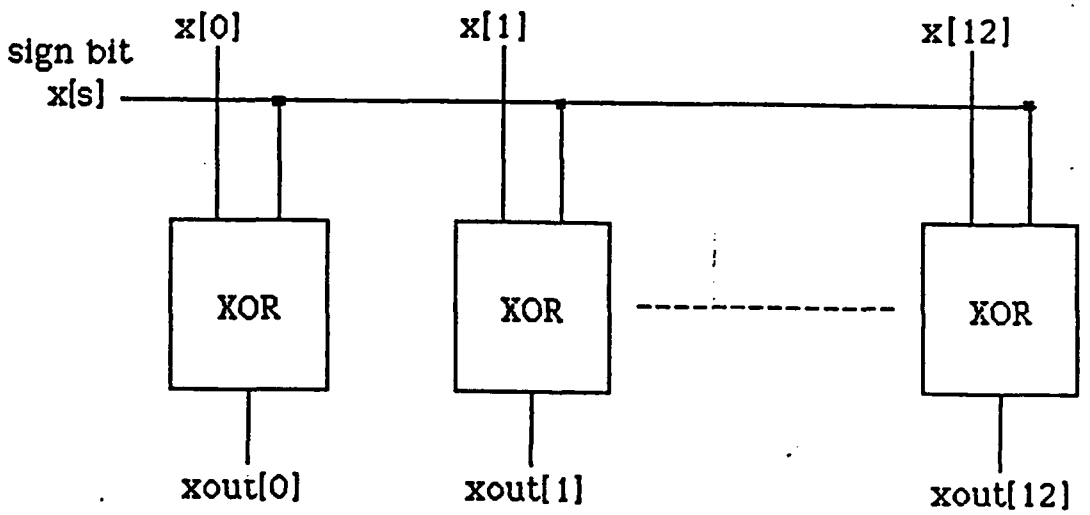


Figure 3.5 Ones' complement to signed magnitude [Ref. 19]

3. Multiplier Block

The multiplier block found in the GENESIL library is an array of half and full adders that provides a parallel multiplier for use in unsigned integer multiplication [Ref. 20]. External circuitry is required for signed multiplication operations. The least significant bits are produced directly from the array of half and full adders, but an external adder is required to complete the partial product addition of the most significant bits. The multiplier and multiplicand widths can vary form 4 to 32 bits, but the multiplier width cannot be greater than the multiplicand width.

4. Signed Magnitude to Ones' Complement Block

The design of this block is very similar to the design of the ones' complement to signed magnitude block. No conversion is required for positive numbers, and the inverse of the magnitude value is used for negative numbers.

5. Adder Block

A full adder from the GENESIL library [Ref. 21] was used for this block. The width of this full adder block can be varied from 1 to 16 bits, so it was necessary to use two blocks configured in a ripple-carry fashion due to the length of the output from the multiplier block. The adder has two data input buses and a carry input line which are added together to produce the data output bus and a carry output line. Figure 3.6 shows the logic design of the adder block.

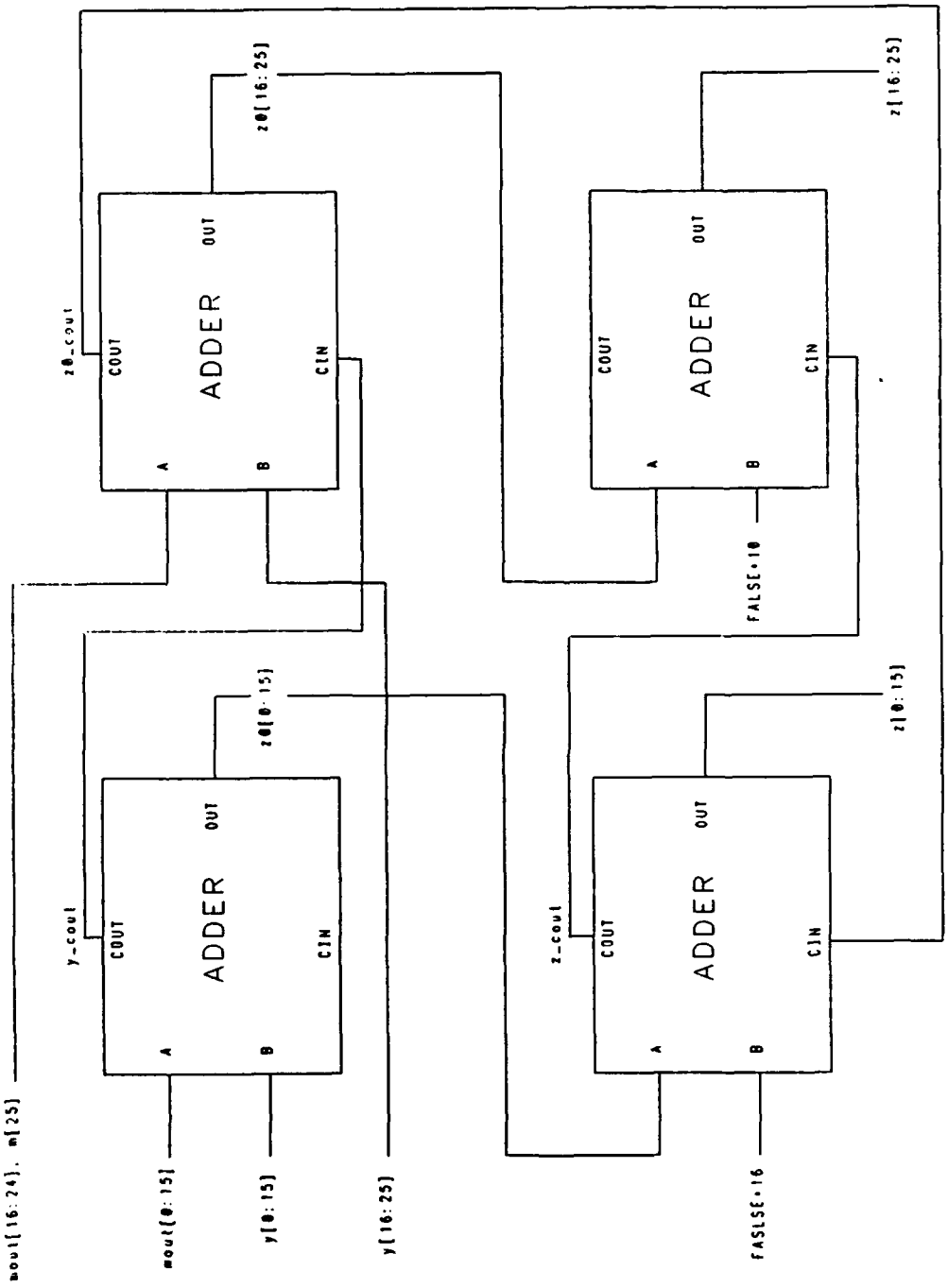


Figure 3.6 Logic design of adder block

6. Overflow Block

The overflow block is designed to detect the presence of an overflow condition for each module on the notch filter chip. Overflow can occur only when both numbers are positive or both numbers are negative. Therefore, overflow can occur only if the sign of the resultant differs from that of the original numbers [Ref. 19:p. 17].

B. IMPLEMENTING RESIDUE CODE INTO THE MULTIPLY-ADD MODULE FOR TESTABILITY

To incorporate design for testability into the multiply-add module of the notch filter, the residue code was chosen because the module performs arithmetic operations. Thus, implementation of residue code to ensure correctness of these arithmetic operations is quite straightforward. The use of a residue code introduces a checking step in the arithmetic operation. The validity of every operand and every result in an operation must be checked. This checking step, therefore, results in a cost which is expressed by an increase in hardware and decrease in speed. To examine the cost of implementing the checking algorithm, a modulo-3 and a modulo-15 low-cost residue code is used for comparison.

As stated earlier, the multiply-add module multiplies a fixed-input (x[t]) by a constant coefficient (a) and adds the result to another input stream (y[t]), thus producing the output stream (z[t]). This can be represented as z[t] = a *

x[t] + y[t]. To check this operation with a residue code, it is necessary to multiply the residue of a by the residue of x using a modulo-m multiplier and then add the resulting residue to the residue of y using a modulo-m adder. Using a comparator, this residue can then be compared to the residue of z. If both residues are the same, the output of the comparator (error) is a logical 0 and no errors have occurred.

Modulo-3 and modulo-15 adders and multipliers were used for this thesis. These adders and multipliers were implemented with the GENESIL Silicon Compiler by using the Programmable Logic Array (PLA) Block. Table 3.1 lists the parameters and options available for the PLA. The Optimizer parameter provides a choice between no optimization of the logic equation or optimization with UC Berkeley Espresso. PLAs are implemented as a two-level sum-of-products expression. As shown in Figure 3.7, the output signals can be expressed as the sum (OR) of several intermediate signals, each of which can be expressed as the product (AND) of several input signals [Ref. 20:p. 6-1]. The specification of the PLA equations is done in a PLA ancillary file with PLAEQ, a PLA programming language with six equation formats: Logic Equation Format, IF Format, Truth Table Format, Switch Actions, Minterm Actions and Finite State Machines. The Truth Table Format is used to specify the modulo-3 and modulo-15 adders and multipliers for this thesis.

Table 3.1 PLA Parameters and Options [Ref. 20]

PARAMETERS	OPTIONS
Inputs	1 - 256
Minterms	1 - 512
Outputs	1 - 256
Timing	Full-Propagate, Input-Latch, Output-Latch, Half-Cycle, Half-Precharged, Full-Precharged
Floorplan	ROM, Decoder
Optimize Logic	Yes, No

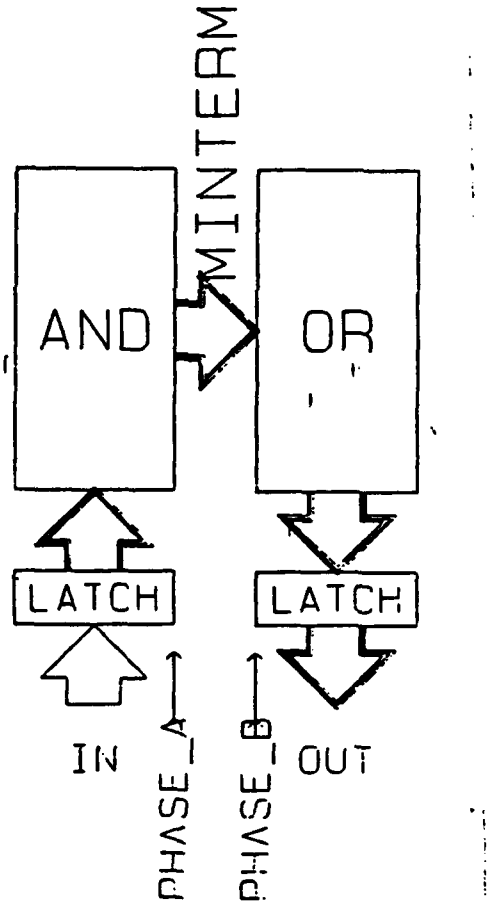


Figure 3.7 GENESIL view of PLA Block [Ref. 20]

C. MODULO-3 LOW-COST RESIDUE CODE IMPLEMENTATION

A block diagram of the modulo-3 low-cost residue code used to verify the ones' complement multiply-add module is shown in Figure 3.8. The checking algorithm consists of seven blocks: residue_generator_a block, residue_generator_x block, mod_3_multiplier block, residue_generator_y block, mod_3_adder block, residue_generator_z block and comparator block.

1. Residue_generator_a Block

The residue of the constant coefficient (a) is generated from its 12 magnitude bits (a[11]...a[0]) by using a modulo-3 low-cost residue code. Remember, low-cost residue codes make encoding easy because division is recast as addition and these codes have a modulus of $m = 2^b - 1$, where $b = 2$ for mod-3. As shown in Figure 3.9, the 12 data bits to be encoded are divided into six groups of two bits. These six groups are then successively added using five, mod-3 adders to form the residue (ra_1, ra_0). This block has a maximum output delay of 24.6 ns and an area of 860.86 sq mils.

2. Residue_generator_x Block

The design of this block is very similar to that of the residue_generator_a block. As shown in Figure 3.10, the 12 magnitude bits (x[11]...x[0]) are divided into six groups of two bits and then successively added using five, mod-3 adders to form the residue (rx_1, rx_0). This block has a maximum output delay of 24.6 ns and an area of 860.86 sq mils.

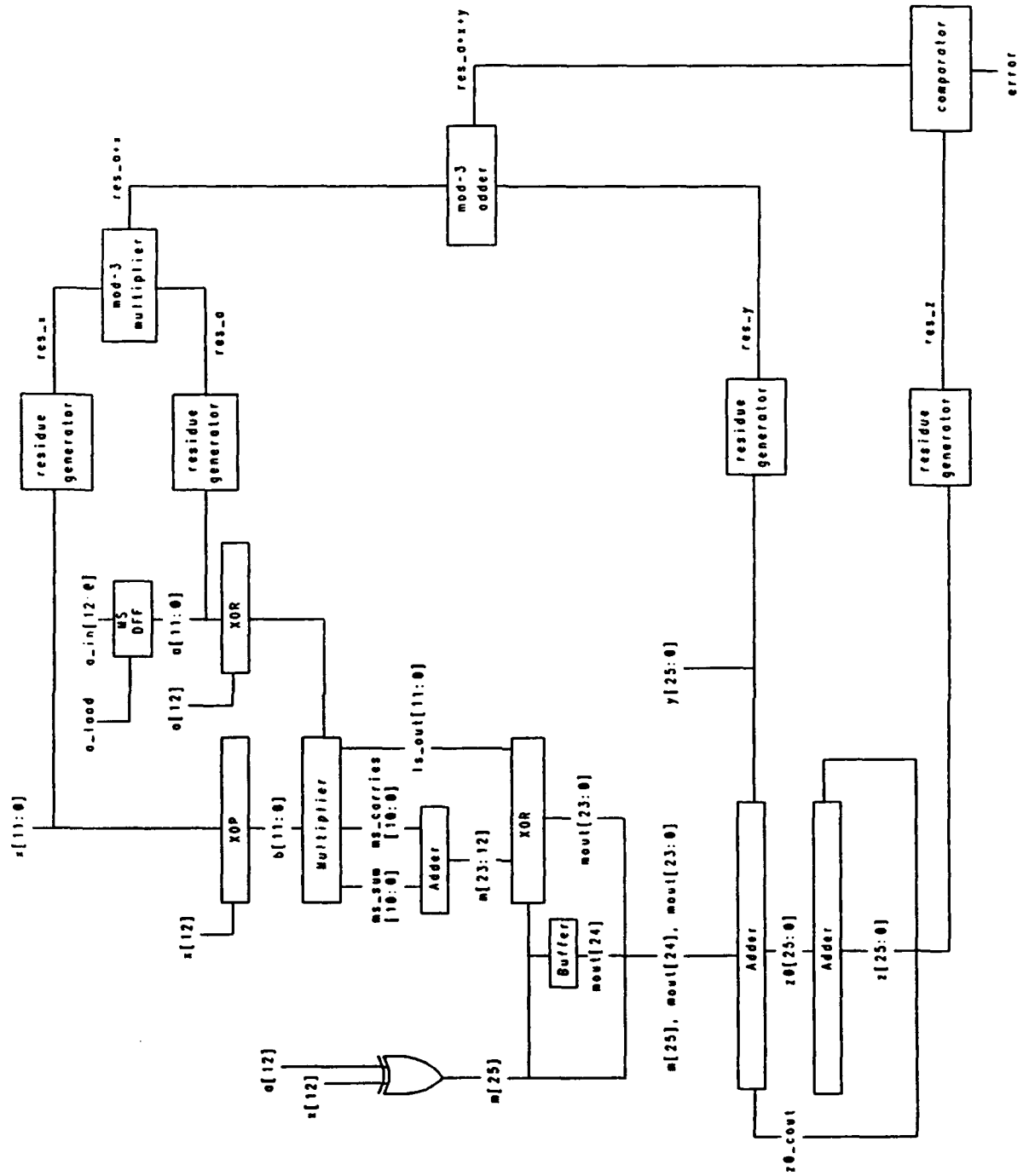


Figure 3.8 Modulo-3 residue code implementation

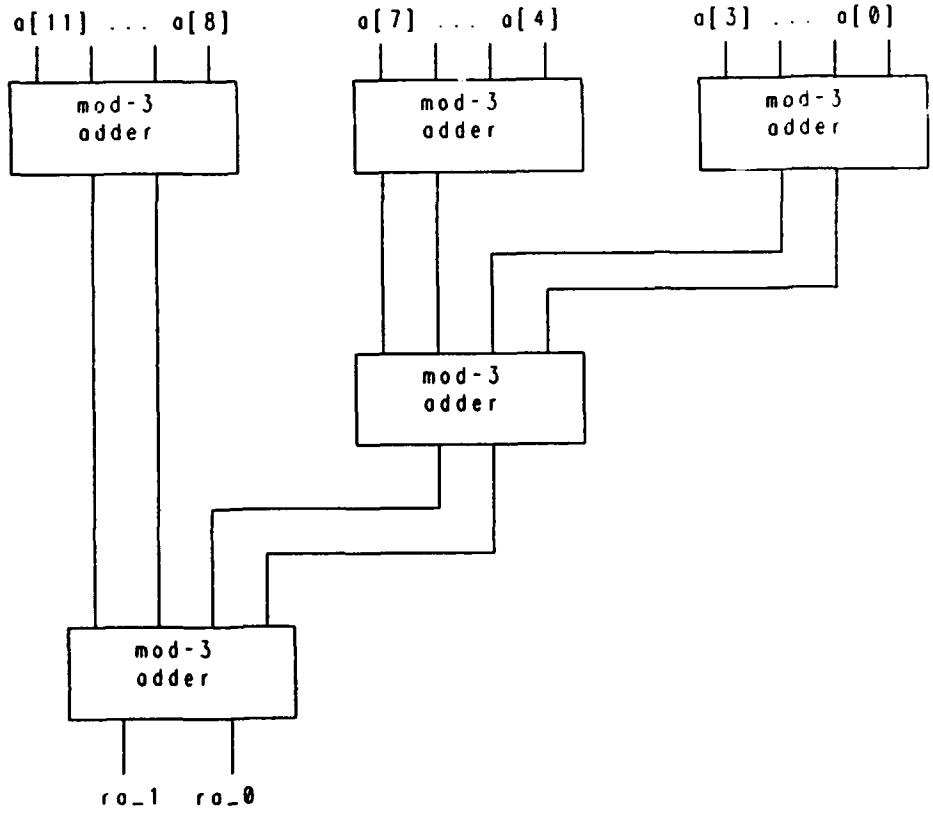


Figure 3.9 Mod-3 residue generation of a

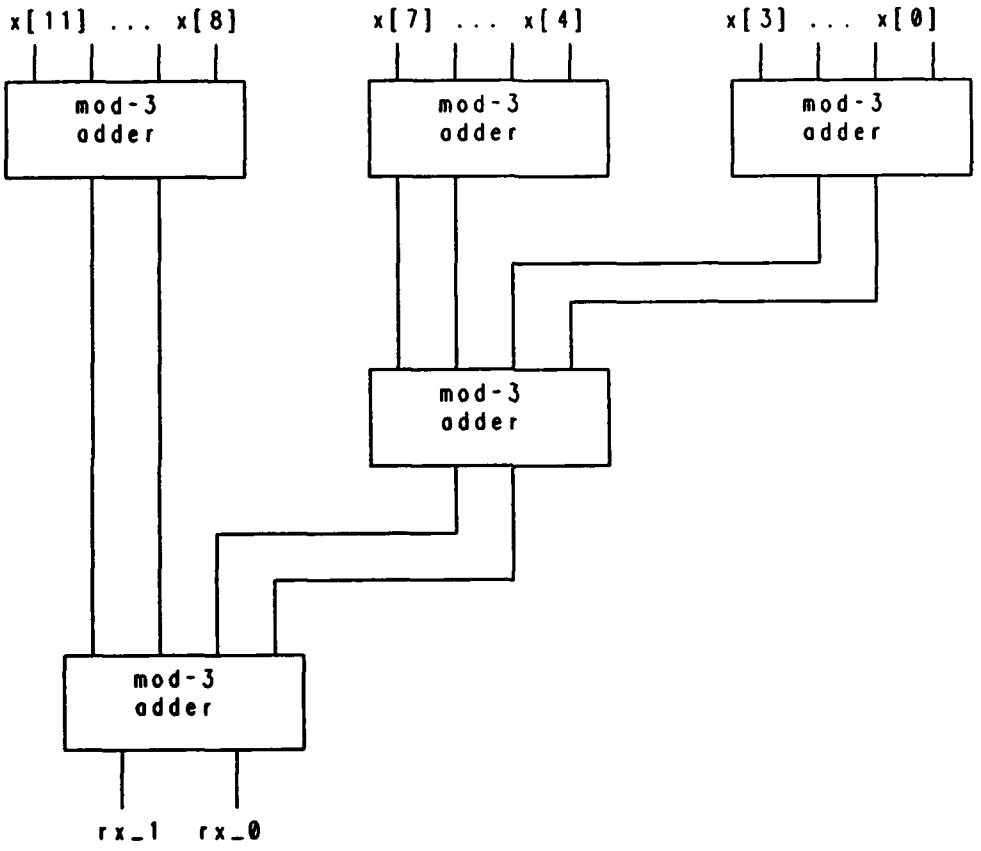


Figure 3.10 Mod-3 residue generation of x[t]

3. Mod_3_multiplier Block

The residue of a * x is generated using the mod_3_multiplier block. The residue of a (ra_1, ra_0) and the residue of x (rx_1, rx_0) are multiplied in a mod-3 fashion to produce the residue of a * x (rs_1_mul, rs_0_mul). This block has a maximum output delay of 5.8 ns and an area of 60.37 sq mils. Figure 3.11 shows a diagram of a mod-3 multiplier and its truth table.

rx_1	rx_0	ra_1	ra_0	rs_1_mul	rs_0_mul
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

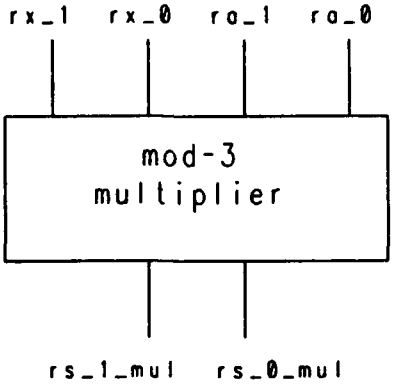


Figure 3.11 Mod-3 residue generation of a * x

4. Residue_generator_y Block

The design of this block is very similar to the design of the residue_generator_a block, but the number of bits used is different. The integer product of a * x produces 26 bits, one sign bit and 25 magnitude bits. So, the input stream (y[t]) must be padded out to 26 bits for addition to the product of a * x. As shown in Figure 3.12, the 26 bits (y[25]...y[0]) are divided into 13 groups of two bits. These 26 bits are then successively added using 12, mod-3 adders to form the residue (ry_1, ry_0). This block has a maximum output delay of 33.6 ns and an area of 4416.00 sq mils.

5. Mod_3_adder Block

The residue of a * x + y is generated using the mod_3_adder block. The residue of a * x (rs_1_mul, rs_0_mul) and the residue of y (ry_1, ry_0) are added in a mod-3 fashion to produce the residue of a * x + y (rs_1, rs_0). This block has a maximum output delay of 8.5 ns and an area of 81.83 sq mils. Figure 3.13 shows a diagram of a mod-3 adder and its truth table.

6. Residue_generator_z Block

This block is similar to the res_gen_y block. As shown in Figure 3.14, the 26 bits (z[25]...z[0]) are divided into 13 groups of two bits and successively added using 12, mod-3 adders to form the residue (rc_1, rc_0). The maximum output delay is 33.6 ns and the area is 4416.00 sq mils.

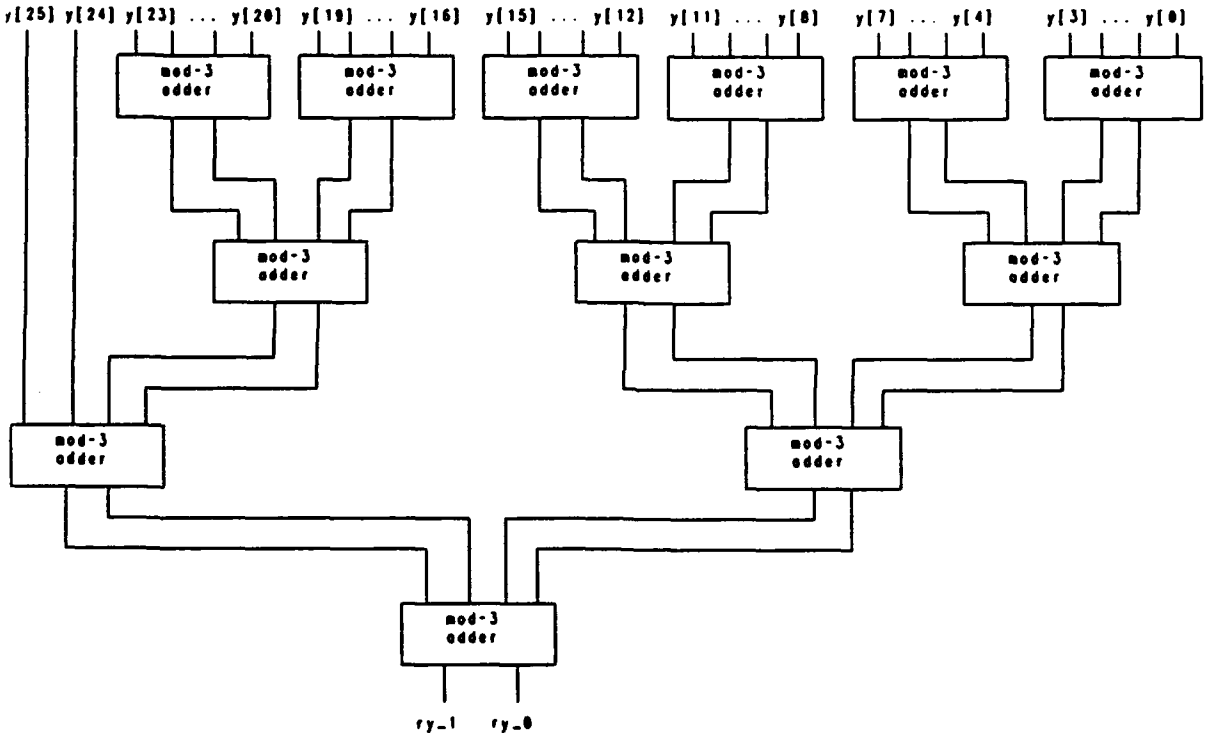


Figure 3.12 Mod-3 residue generation of y[t]

rs_1_mul	rs_0_mul	ry_1	ry_0	rs_1	rs_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	0	0

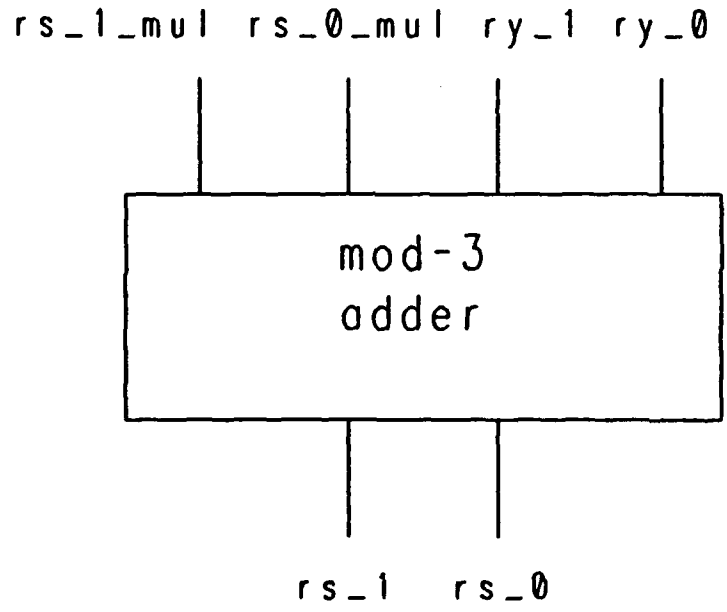


Figure 3.13 Mod-3 residue generation of a * x + y

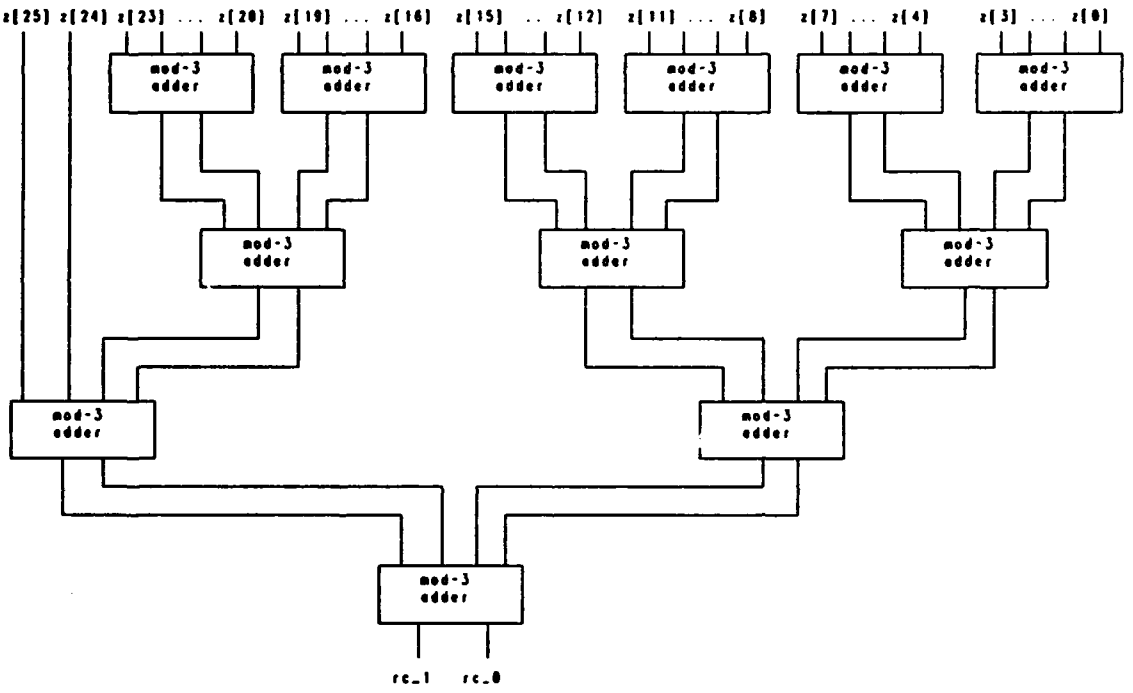


Figure 3.14 Mod-3 residue generation of z[t]

7. Comparator Block

The comparator block is used to compare the residue of a * x + y (rs_1, rs_0) to the residue of z (rc_1, rc_0) for detecting errors. The block uses two XOR gates to compare the residues. For example, XOR_1 has inputs of rs_1 and rc_1 and as long as these inputs have the same logic level, the output of XOR_1 will be logic zero. An OR gate is then used to check the outputs of XOR_1 and XOR_0. If both of these outputs are logic zero, then the output of the OR gate (error) will be logic zero, indicating no errors. However, if any XOR gate has two different input values, that XOR gate will produce a logic one output, causing the OR gate to correspondingly produce a logic one output on the error line. This logic one output from the comparator indicates there is an error, but it does not indicate whether the error occurred in the arithmetic operation or in the checking step. This block has a maximum output delay of 2.5 ns and and area of 11.87 sq mils. Figure 3.15 shows the comparator block.

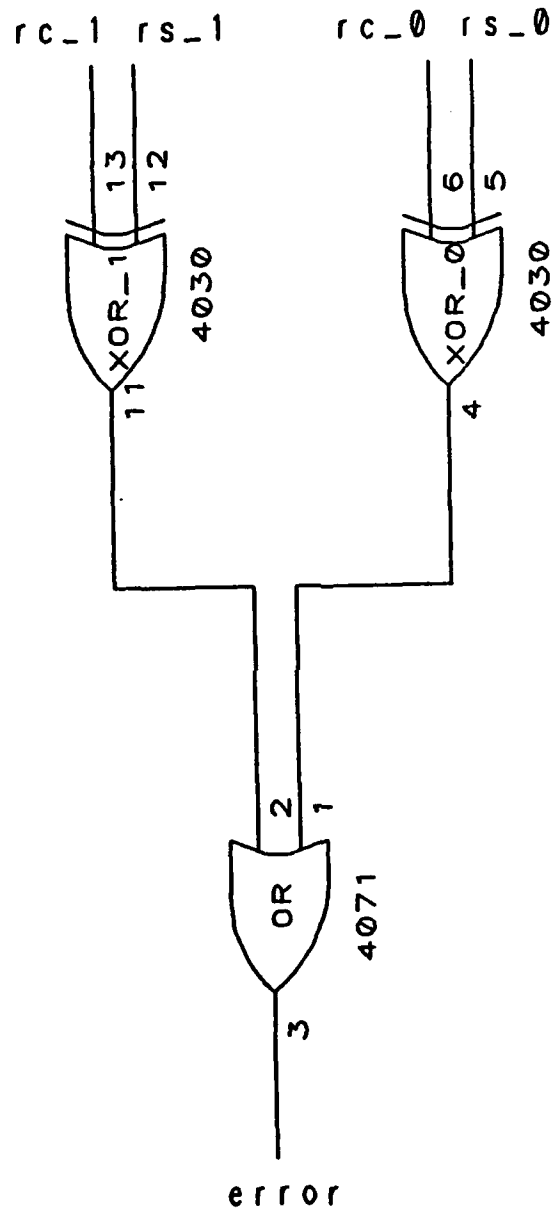


Figure 3.15 Mod-3 comparator block

D. MODULO-15 LOW-COST RESIDUE CODE IMPLEMENTATION

A block diagram of the modulo-15 low-cost residue code used to verify the ones' complement multiply-add module is shown in Figure 3.16. The design of the checking algorithm is very similar to that used in mod-3 and consists of seven blocks: residue_gen_a, res_gen_x, mod_15_mult, residue_gen_y, mod-15_adder, residue_gen_z and comparator block.

1. Residue_generator_a Block

The residue of the constant coefficient (a) is generated from its 12 magnitude bits (a[11]...a[0]) by using a modulo-15 low-cost residue code. The low-cost residue code has a modulus of $m = 2^b - 1$, where $b = 4$ for mod-15. As shown in Figure 3.17, the 12 data bits to be encoded are divided into three groups of four bits and successively added using two, mod-15 adders to form the residue (ra_3, ra_2, ra_1, ra_0). This block has a maximum output delay of 60.5 ns and an area of 1666.56 sq mils.

2. Residue_generator_x Block

The design of this block is similar to that of the residue_generator_a block. As shown in Figure 3.18, the 12 magnitude bits (x[11]...x[0]) are divided into three groups of four bits and successively added using two, mod-15 adders to form the residue (rx_3, rx_2, rx_1, rx_0). This block has a maximum output delay of 60.5 ns and an area of 1666.56 sq mils.

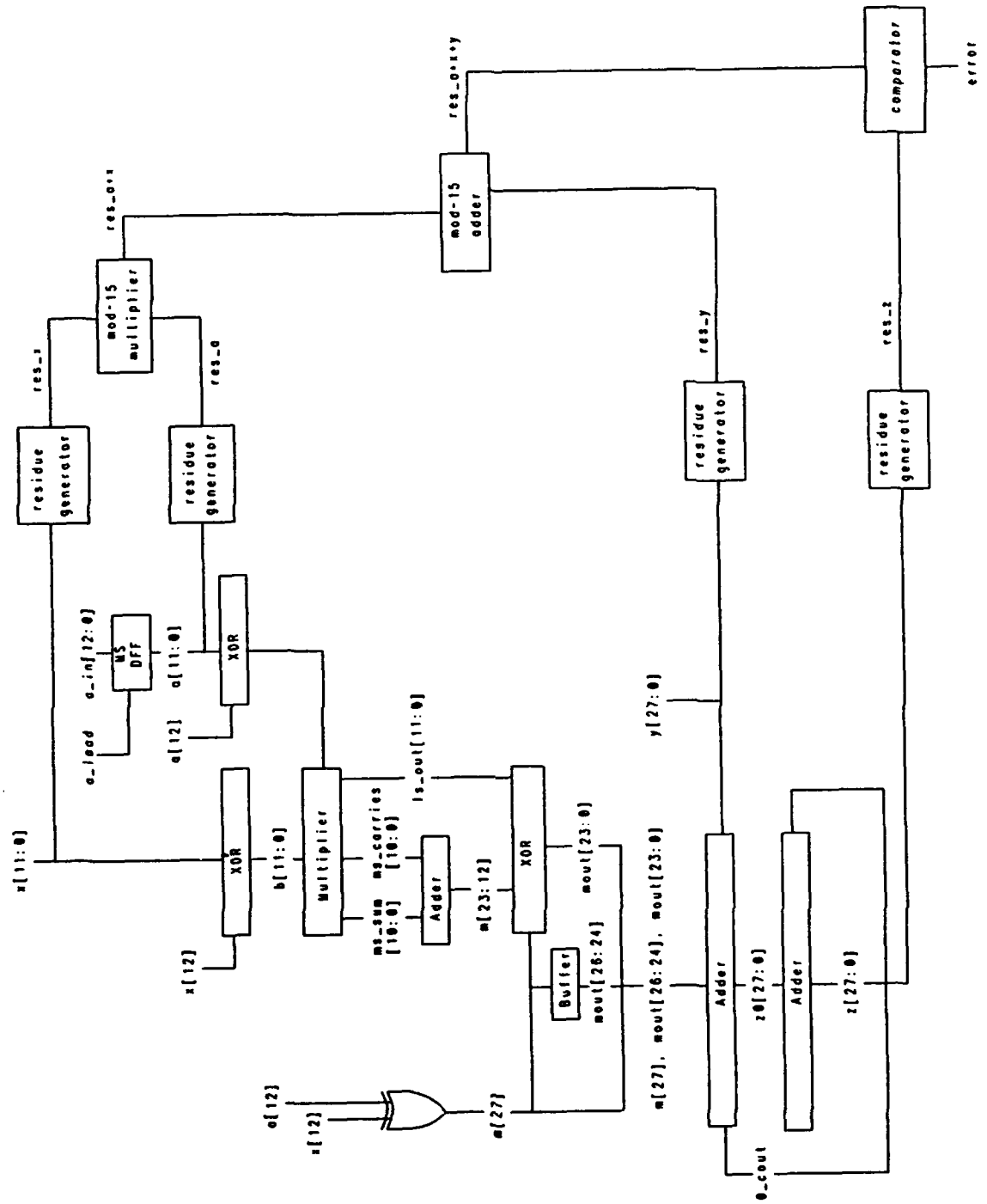


Figure 3.16 Modulo-15 residue code implementation

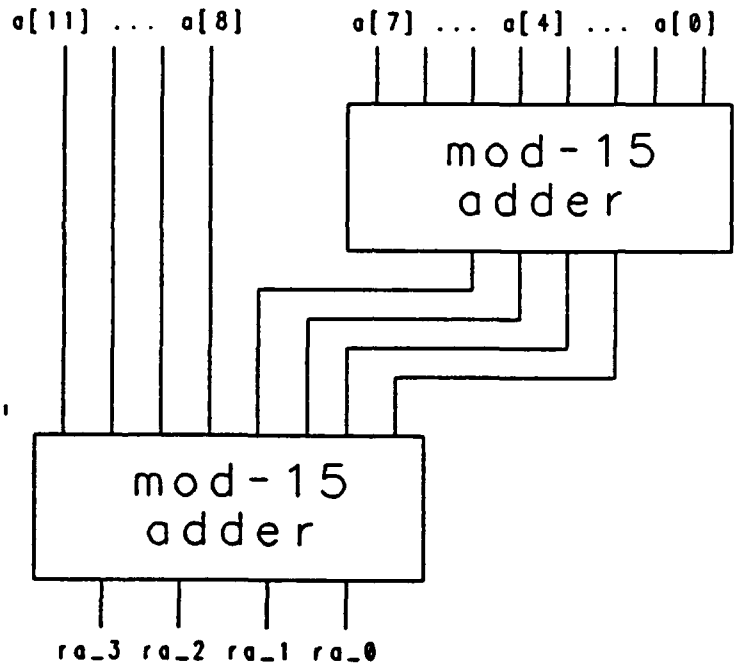


Figure 3.17 Mod-15 residue generation of a

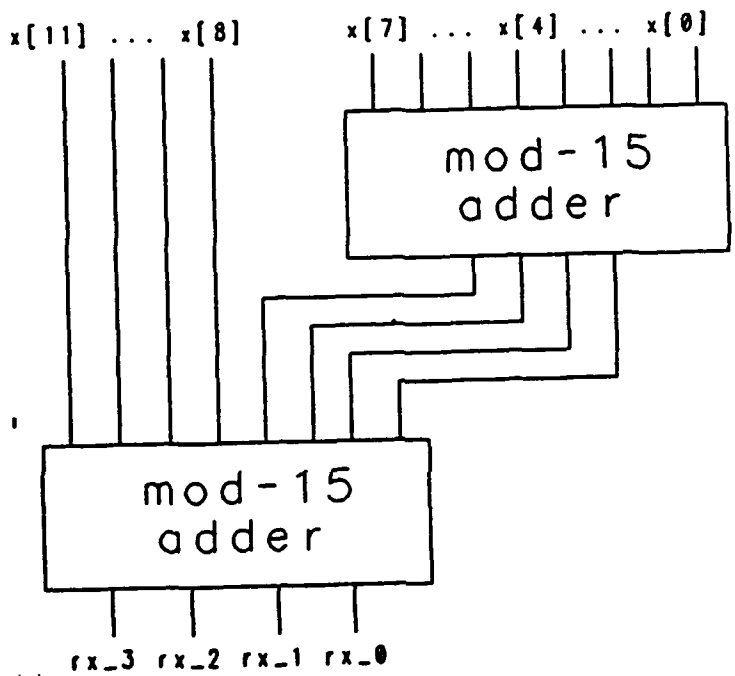


Figure 3.18 Mod-15 residue generation of x[t]

3. Mod_15_multiplier Block

The residue of a * x is generated using the mod_15_multiplier block. The residue of **a** (ra_3, ra_2, ra_1, ra_0) and the residue of **x** (rx_3, rx_2, rx_1, rx_0) are multiplied in modulo-15 fashion to produce the residue of a*x (ax_3, ax_2, ax_1, ax_0). The maximum output delay is 34.1 ns and the area is 674.30 sq mils. Figure 3.19 and Table 3.2 show a diagram of a mod-15 multiplier and its truth table.

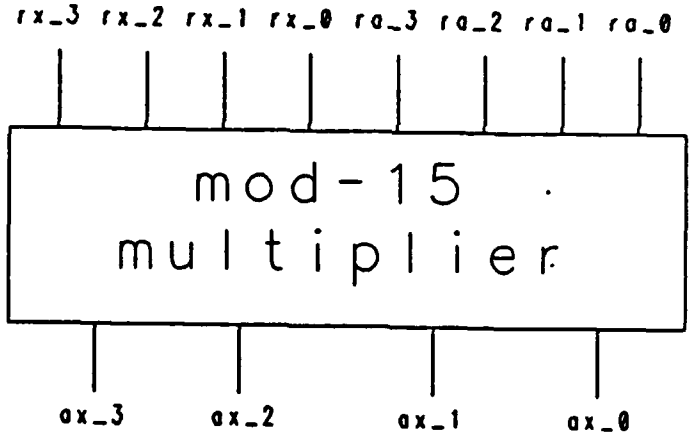


Figure 3.19 Mod-15 residue generation of a * x

Table 3.2 Truth table for mod-15 multiplier

rx_3	rx_2	rx_1	rx_0	ra_3	ra_2	ra_1	ra_0	ax_3	ax_2	ax_1	ax_0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
	•	•		•	•				•	•	
	•	•		•	•				•	•	
0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	1	1	1	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	0
	•	•		•	•				•	•	
	•	•		•	•				•	•	
0	0	0	1	1	1	1	0	1	1	1	0
0	0	0	1	1	1	1	1	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	0	0	1	0	0
	•	•		•	•				•	•	
	•	•		•	•				•	•	
0	0	1	0	1	1	1	0	1	1	0	1
0	0	1	0	1	1	1	1	0	0	0	0
	•	•		•	•				•	•	
	•	•		•	•				•	•	
1	1	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	1	1	1	1	0
1	1	1	0	0	0	1	0	1	1	0	1
	•	•		•	•				•	•	
	•	•		•	•				•	•	
1	1	1	0	1	1	1	0	0	0	0	1
1	1	1	0	1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0	0
	•	•		•	•				•	•	
	•	•		•	•				•	•	
1	1	1	1	1	1	1	1	0	0	0	0

4. Residue_generator_y Block

The design of this block is similar to the design of the mod-3 res_gen_y block, but the input stream (y[t]) must be padded out to 28 bits to accommodate the use of mod-15 adders. As shown in Figure 3.20, the 28 bits (y[27]...y[0]) are divided into seven groups of four bits. These 28 bits are then successively added using six, mod-15 adders to form the residue (ry_3, ry_2, ry_1, ry_0). The maximum output delay is 90.5 ns and the area is 4962.72 sq mils.

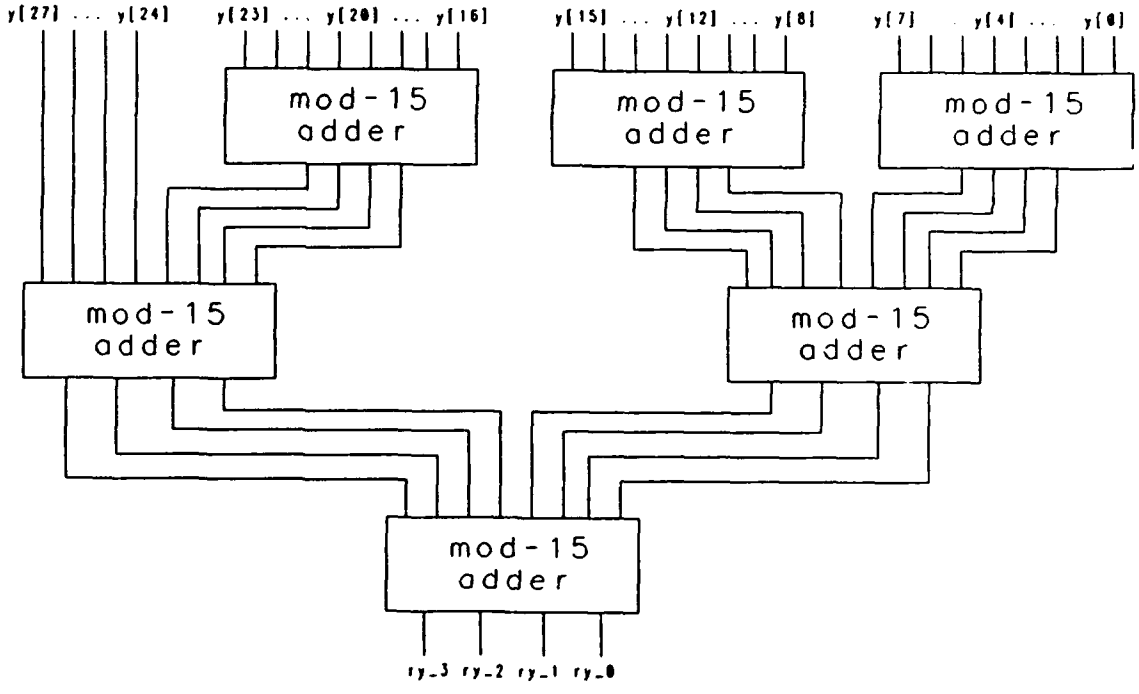


Figure 3.20 Mod-15 residue generation of y[t]

5. Mod_15_adder Block

The residue of a * x + y is generated using the mod_15_adder block. As shown in Figure 3.21, the residue of a * x (ax_3, ax_2, ax_1, ax_0) and the residue of y (ry_3, ry_2, ry_1, ry_0) are added in mod-15 fashion to produce the residue of a * x + y (rs_3, rs_2, rs_1, rs_0). The maximum output delay of this block is 30.5 ns and the silicon area is 714.02 sq mils. Figure 3.21 and Table 3.3 show a diagram of a mod-15 adder and its truth table.

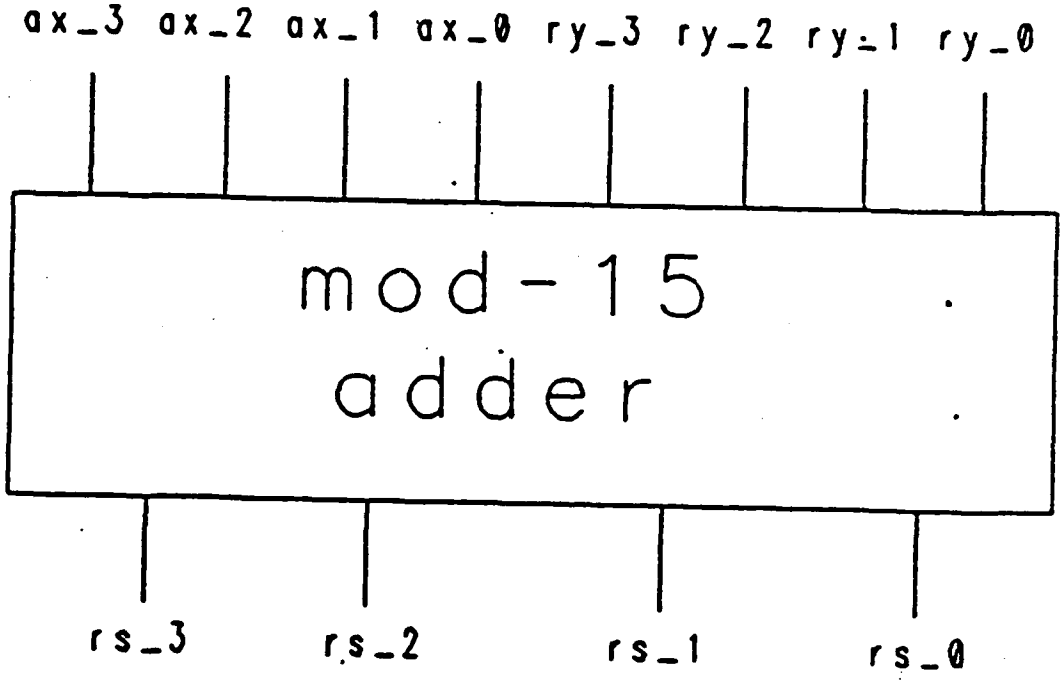


Figure 3.21 Mod-15 residue generation of a * x + y

Table 3.3 Truth table for mod-15 adder

ax_3	ax_2	ax_1	ax_0	ry_3	ry_2	ry_1	ry_0	rs_3	rs_2	rs_1	rs_0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	0	0	0	1	0
	.	.		.	.				.	.	
	.	.		.	.				.	.	
0	0	0	0	1	1	1	0	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0	0	1	0
0	0	0	1	0	0	1	0	0	0	1	1
	.	.		.	.				.	.	
	.	.		.	.				.	.	
0	0	0	1	1	1	1	0	0	0	0	0
0	0	0	1	1	1	1	1	0	0	0	1
0	0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	1	0	0	1	1
0	0	1	0	0	0	1	0	0	1	0	0
	.	.		.	.				.	.	
	.	.		.	.				.	.	
0	0	1	0	1	1	1	0	0	0	0	1
0	0	1	0	1	1	1	1	0	0	1	0
	.	.		.	.				.	.	
	.	.		.	.				.	.	
1	1	1	0	0	0	0	0	1	1	1	0
1	1	1	0	0	0	0	1	0	0	0	0
1	1	1	0	0	0	1	0	0	0	0	1
	.	.		.	.				.	.	
	.	.		.	.				.	.	
1	1	1	0	1	1	1	0	1	1	0	1
1	1	1	0	1	1	1	1	1	1	1	0
1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1	0	0	0	1
1	1	1	1	0	0	1	0	0	0	1	0
	.	.		.	.				.	.	
	.	.		.	.				.	.	
1	1	1	1	1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	1	0	0	0	0

6. Residue_generator_z Block

The design of this block is similar to the design of the residue_generator_y block. The 28 bits (z[27]...z[0]) are divided into seven groups of four bits. These 28 bits are then successively added using six, mod-15 adders to form the residue (rc_3, rc_2, rc_1, rc_0). This block has a maximum output delay of 90.5 ns and an area of 4962.72 sq mils. Figure 3.22 shows the residue generation of z[t].

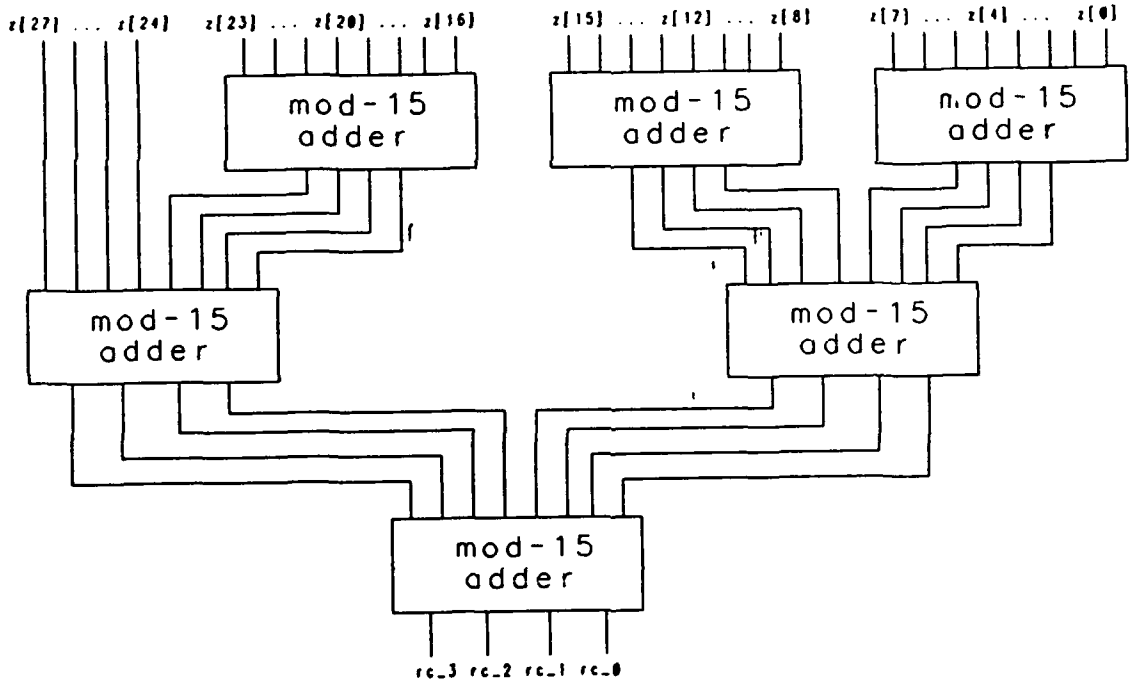


Figure 3.22 Mod-15 residue generation of z[t]

7. Comparator Block

The comparator block is used to compare the residue of a * x + y (rs_3, rs_2, rs_1, rs_0) to the residue of z (rc_3, rc_2, rc_1, rc_0) for errors. As shown in Figure 3.23, this block uses four XOR gates to compare the residues and one, four input OR gate to check the output of the XOR gates (cmp_3, cmp_2, cmp_1, cmp_0). If there are no errors, the output of the OR gate (error) will be a logic zero. This block has a maximum output delay of 2.7 ns and an area of 22.35 sq mils.

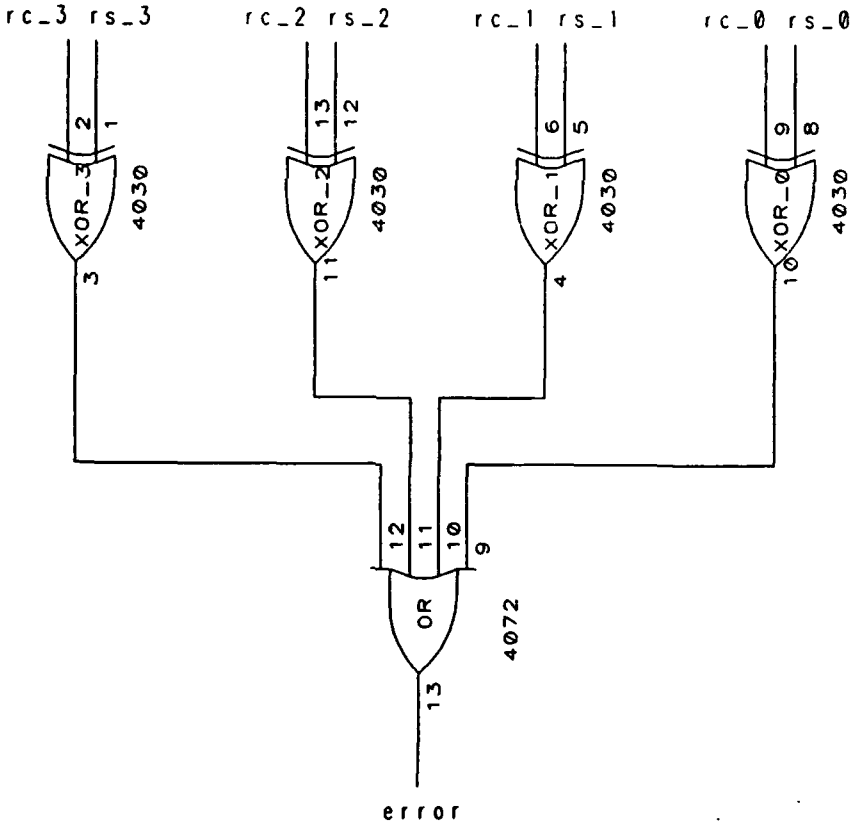


Figure 3.23 Mod-15 comparator block

E. SIMULATION

GENESIL'S ability to perform simulation provides a process by which outputs can be checked against a sequence of inputs to verify that the design is logically correct. The design can by simulated in a manual interactive mode or in an automatic control mode. The manual mode requires that the user specify each input by binding the input pins to a logic zero or logic one, manually advance the clocking signals and then verify each output individually.

The manual mode was used to test and simulate the designs for this thesis. Various combinations of binary integers were placed on the input signal buses, the system clock cycled and the test results were observed on the output signal buses. Figures 3.25 through 3.28 and Figures 3.29 through 3.32 show a sample of simulation tests run for mod-3 and mod-15 implementations, respectively. As shown in Figure 3.33, a stuck-at-zero fault was induced at Bit 0 in the one_to_sm block to demonstrate the ability of the checking algorithm to detect and error. Figures 3.34 through 3.37 and Figures 3.38 through 3.41 show a sample of simulation tests run for the detected s-a-0 fault using the mod-3 and mod-15 implementations, respectively.

```
MULTIPLE_SIGS
x[12:0]
0b1110011110001
a[12:0]
0b0011000000101
y[25:0]
0b1111111111111111100100001010
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLLLLLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 1111101101100101010111000100
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHHHHHHHHHHHHHHHLLHLLLLLHLHL
) port 17 I x[12:0]  to  NC*13 = HHHLLHHHHLLLH
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 0
) port 23 O rs_1  to  NC = 0
) port 25 O error  to  NC = 0
) port 27 O rc_0  to  NC = 0
```

Figure 3.25 Simulation results of multiply-add module(mod-3)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1010011110110
a[12:0]
0b0011011100101

y[25:0]
0b11011001111010101110001010
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLHHHLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 11000110111001011001111101
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHLHHLLHHHHLHLHLHHHLLLHLHL
) port 17 I x[12:0]  to  NC*13 = HLHLLHHHHLHHL
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 1
) port 23 O rs_1  to  NC = 1
) port 25 O error  to  NC = 0
) port 27 O rc_0  to  NC = 0
```

Figure 3.26 Simulation results of multiply-add module(mod-3)

```
MULTIPLE_SIGS
x[12:0]
0b1001110001111
a[12:0]
0b1100111001101
y[25:0]
0b1111111111001011001110010l
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = HHLLHHHLLHHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 00010011000011101011000110
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHHHHHHHHHLLHLHHLLHHHLLHLH
) port 17 I x[12:0]  to  NC*13 = HLLHHHLLLHHHH
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 1
) port 23 O rs_1  to  NC = 1
) port 25 O error  to  NC = 0
) port 27 O rc_0  to  NC = 0
BACK
EXIT_SIM
CONFIRM
SELECT_OBJECT
UP
DOWN
```

Figure 3.27 Simulation results of multiply-add module(mod-3)

```
MULTIPLE_SIGS
x[12:0]
0b1010111011101
a[12:0]
0b0110100101011
y[25:0]
0b1111111111001100110000 1101
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LHHLHLLHLHLHH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 11011110011100010101010111
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHHHHHHHHHLLHHLLHHLLLLHHLH
) port 17 I x[12:0]  to  NC*13 = HLHLHHHLHHHLH
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 0
) port 23 O rs_1  to  NC = 0
) port 25 O error  to  NC = 0
) port 27 O rc_0  to  NC = 0
```

Figure 3.28 Simulation results of multiply-add module(mod-3)

```
x[12:0]
0b1110011110001
a[12:0]
0b0011000000101
y[27:0]
0b1111111111111111100100001010
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLLLLLLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 1111111011011001010111000100
) port 13 I y[27:0]  to  NC*28 = HHHHHHHHHHHHHHHHHHLL HLLLLLHLHL
) port 15 I x[12:0]  to  NC*13 = HHHLLHHHHLLLH
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 0
) port 21 O rc_2  to  NC = 1
) port 23 O rc_3  to  NC = 1
) port 25 O rc_0  to  NC = 0
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 0
) port 31 O rs_2  to  NC = 1
) port 33 O rs_3  to  NC = 1
) port 35 O error  to  NC = 0
```

Figure 3.29 Simulation results of multiply-add module(mod-15)

```
MULTIPLE_SIGS
x[12:0]
0b1010011110110
a[12:0]
0b0011011100101
y[27:0]
0b1111011001111010101110001010
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLHHHLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 1111000110111001011001111101
) port 13 I y[27:0]  to  NC*28 = HHHHLHHLLHHHHLHLHLHHHLLLHLHL
) port 15 I x[12:0]  to  NC*13 = HLHLLHHHHLHHL
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 1
) port 21 O rc_2  to  NC = 0
) port 23 O rc_3  to  NC = 0
) port 25 O rc_0  to  NC = 0
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 1
) port 31 O rs_2  to  NC = 0
) port 33 O rs_3  to  NC = 0
) port 35 O error  to  NC = 0
```

Figure 3.30 Simulation results of multiply-add module(mod-15)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1001110001111
a[12:0]
0b1100111001101
y[27:0]
0b1111111111110010110011100101
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = HHLLHHHLLHHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 0000010011000011101011000110
) port 13 I y[27:0]  to  NC*28 = HHHHHHHHHHHHLLHLHHLLHHHLLHLH
) port 15 I x[12:0]  to  NC*13 = HLLHHHLLLHHHH
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 1
) port 21 O rc_2  to  NC = 0
) port 23 O rc_3  to  NC = 0
) port 25 O rc_0  to  NC = 0
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 1
) port 31 O rs_2  to  NC = 0
) port 33 O rs_3  to  NC = 0
) port 35 O error  to  NC = 0
BACK
EXIT_SIM
CONFIRM
EXIT_GENESIL
CONFIRM
```

Figure 3.31 Simulation results of multiply-add module(mod-15)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1010111011101
a[12:0]
0b0110100101011
y[27:0]
0b1111111111110011001100001101
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LHHLHLLLHLHLHH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 11110111100111000101010101111
) port 13 I y[27:0]  to  NC*28 = HHHHHHHHHHHHHLLHHLLHHLLLLHHLH
) port 15 I x[12:0]  to  NC*13 = HLHLHHHLHHHLH
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 0
) port 21 O rc_2  to  NC = 0
) port 23 O rc_3  to  NC = 0
) port 25 O rc_0  to  NC = 0
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 0
) port 31 O rs_2  to  NC = 0
) port 33 O rs_3  to  NC = 0
) port 35 O error  to  NC = 0
```

Figure 3.32 Simulation results of multiply-add module(mod-15)

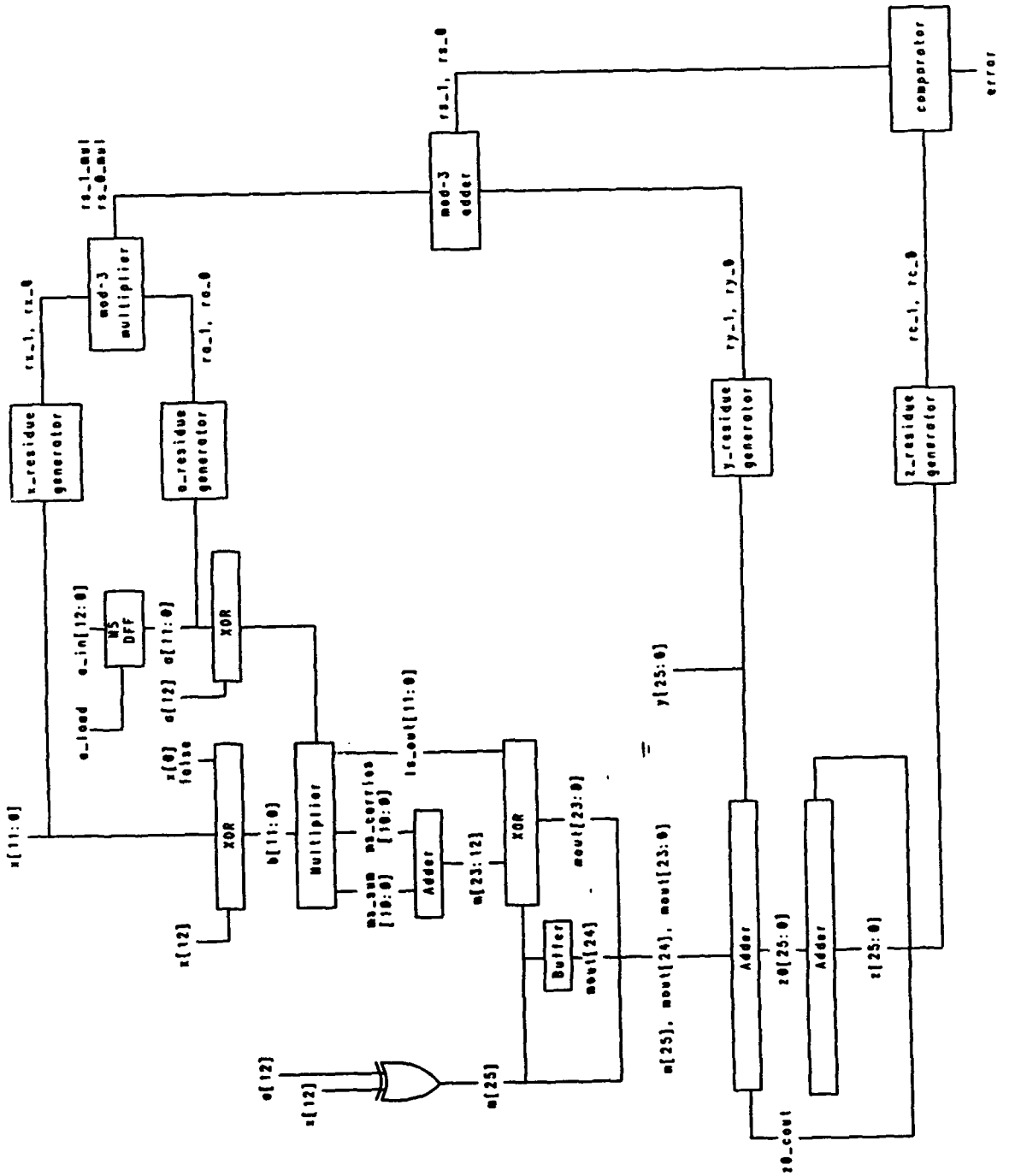


Figure 3.33 Residue code implementation with x[0] s-a-0 fault

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1110011110001
a[12:0]
0b0011000000101
y[25:0]
0b11111111111111100100001010
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLLLLLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 11111011011000111110111111
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHHHHHHHHHHHHHHHLLHLLLLLHLHL
) port 17 I x[12:0]  to  NC*13 = HHHLLHHHHLLLH
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 0
) port 23 O rs_1  to  NC = 0
) port 25 O error  to  NC = 1
) port 27 O rc_0  to  NC = 1
```

Figure 3.34 s-a-0 fault simulation (mod-3)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1010011110110
a[12:0]
0b0011011100101
y[25:0]
0b11011001111010101110001010
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLHHHLLHLH
) port 7 CI phase_b  to  NC = 0

) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 11000110111001011001111101
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHLHHLLHHHHLHLHLHHHLLLHLHL
) port 17 I x[12:0]  to  NC*13 = HLHLLHHHHLHHL
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 1
) port 23 O rs_1  to  NC = 1
) port 25 O error  to  NC = 0
) port 27 O rc_0  to  NC = 0
```

Figure 3.35 s-a-0 fault simulation (mod-3)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1001110001111
a[12:0]
0b1100111001101
y[25:0]
0b11111111110010110011100101
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = HHLLHHHLLHHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 00010011000100000011111000
) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHHHHHHHHHLLHLHHLLHHHLLHLH
) port 17 I x[12:0]  to  NC*13 = HLLHHHLLLHHHH
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 0
) port 23 O rs_1  to  NC = 1
) port 25 O error  to  NC = 1
) port 27 O rc_0  to  NC = 1
```

Figure 3.36 s-a-0 fault simulation (mod-3)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1010111011101
a[12:0]
0b0110100101011
y[25:0]
0b11111111110011001100001101
BACK
CYCLE
1
pi
)   is of type  module  with  28  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LHHLHLLHLHLHH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[25:0]  to  NC*26 = 11011110011011100000101100

) port 13 O rs_0  to  NC = 0
) port 15 I y[25:0]  to  NC*26 = HHHHHHHHHHLLHHLLHHLLLLHHLH
) port 17 I x[12:0]  to  NC*13 = HLHLHHHLHHHLH
) port 19 O overflow  to  NC = i
) port 21 O rc_1  to  NC = 0
) port 23 O rs_1  to  NC = 0
) port 25 O error  to  NC = 1
) port 27 O rc_0  to  NC = 1
BACK
EXIT_SIM
CONFIRM
EXIT_GENESIL
```

Figure 3.37 s-a-0 fault simulation (mod-3)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1110011110001
a[12:0]
0b0011000000101
y[27:0]
0b11111111111111111100100001010
BACK

CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLLLLLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 1111111011011000111110111111
) port 13 I y[27:0]  to  NC*28 = HHHHHHHHHHHHHHHHHHLLHLLLLLHLHL
) port 15 I x[12:0]  to  NC*13 = HHHLLHHHHLLLH
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 0
) port 21 O rc_2  to  NC = 0
) port 23 O rc_3  to  NC = 0
) port 25 O rc_0  to  NC = 1
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 0
) port 31 O rs_2  to  NC = 1
) port 33 O rs_3  to  NC = 1
) port 35 O error  to  NC = 1
```

Figure 3.38 s-a-0 fault simulation (mod-15)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1010011110110
a[12:0]
0b0011011100101
y[27:0]
0b1111011001111010101110001010
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LLHHLHHHLLHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 1111000110111001011001111101
) port 13 I y[27:0]  to  NC*28 = HHHHLHHLLHHHHLHLHLHHHLLLHLHL
) port 15 I x[12:0]  to  NC*13 = HLHLLHHHHLHHL
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 1
) port 21 O rc_2  to  NC = 0
) port 23 O rc_3  to  NC = 0
) port 25 O rc_0  to  NC = 0
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 1
) port 31 O rs_2  to  NC = 0
) port 33 O rs_3  to  NC = 0
) port 35 O error  to  NC = 0
```

Figure 3.39 s-a-0 fault simulation (mod-15)

```
BIND
MULTIPLE_SIGS
x[12:0]

0b1001110001111
a[12:0]
0b1100111001101
y[27:0]
0b1111111111110010110011100101
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = HHLLHHHLLHHLH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 0000010011000100000011111000
) port 13 I y[27:0]  to  NC*28 = HHHHHHHHHHHHLLHLHHLLHHHLLHLH
) port 15 I x[12:0]  to  NC*13 = HLLHHHLLLHHHH
) port 17 O overflow  to  NC = i
) port 19 O rc_1  to  NC = 0
) port 21 O rc_2  to  NC = 1
) port 23 O rc_3  to  NC = 1
) port 25 O rc_0  to  NC = 1
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 1
) port 31 O rs_2  to  NC = 0
) port 33 O rs_3  to  NC = 0
) port 35 O error  to  NC = 1
```

Figure 3.40 s-a-0 fault simulation (mod-15)

```
BIND
MULTIPLE_SIGS
x[12:0]
0b1010111011101
a[12:0]
0b0110100101011
y[27:0]
0b1111111111110011001100001101
BACK
CYCLE
1
pi
)   is of type  module  with  36  ports
) port 1 I TRUE  to  NC = H
) port 3 I FALSE  to  NC = L
) port 5 I a[12:0]  to  NC*13 = LHHLHLLLHLHLHH
) port 7 CI phase_b  to  NC = 0
) port 9 CI phase_a  to  NC = 1
) port 11 O z[27:0]  to  NC*28 = 1111011110011011100000101100
) port 13 I y[27:0]  to  NC*28 = HHHHHHHHHHHHHLLHHLLHHLLLLHHLH
) port 15 I x[12:0]  to  NC*13 = HLHLHHHLHHHLH
) port 17 O overflow  to  NC = 1
) port 19 O rc_1  to  NC = 0
) port 21 O rc_2  to  NC = 1
) port 23 O rc_3  to  NC = 0
) port 25 O rc_0  to  NC = 0
) port 27 O rs_0  to  NC = 0
) port 29 O rs_1  to  NC = 0

) port 31 O rs_2  to  NC = 0
) port 33 O rs_3  to  NC = 0
) port 35 O error  to  NC = 1
BACK
EXIT_SIM
CONFIRM
EXIT_GENESIL
CONFIRM
```

Figure 3.41 s-a-0 fault simulation (mod-15)

IV CONCLUSIONS

A. SUMMARY

The main goal of this thesis is to describe the need for including design for testability in a VLSI chip design and to provide information on implementing a DFT strategy using the GENESIL Silicon Compiler. Specifically, this thesis describes the implementation of residue code as a checking algorithm for testing the multiply-add module of a notch filter.

The material in Chapter I provides background information on testability issues, fault models and the Genesil Silicon Compiler. Chapter II discusses design for testability, in general, and describes two structured techniques: Scan Design methods and Built-in Self Test approaches, including residue code. Chapter III describes the basic design of a second-order Infinite Impulse Response notch filter and includes a complete functional description of the multiply-add module. This chapter also describes the methodology used to implement residue code with the GENESIL Silicon Compiler for testing the multiply-add module.

The results of this thesis indicate that, in fact, a residue code can successfully be implemented as a design for testability strategy using GENESIL to test the multiply-add module of a notch filter. However, there is a cost in terms

of increased hardware and decreased performance that accompanies the checking algorithm. The modulo-3 implementation has a maximum output delay of 122.4 ns and a silicon area of 92,376.90 sq mils which represents an increase in area of 81,067.34 sq mils and a decrease in performance of 59.9 ns. The modulo-15 implementation has a maximum output delay of 182.4 ns and a silicon area of 112,786.52 sq mils which represents an increase in area of 101,476.96 sq mils and a decrease in performance of 119.9 ns. The modulo-15 implementation is more costly due to the increased complexity of the Programmable Logic Array Blocks used for the residue generation.

B. RECOMMENDATIONS

The following are recommendations for further study:

1. Implement a residue code that provides single-error-correcting capability.
2. Implement an inverse residue code which is a variant of the residue code specifically designed for fault-detection of repeated-use faults. Repeated use faults are particularly difficult to detect because latter effects of the fault can cancel the previous effects, rendering the fault undetectable.
3. Implement a multiresidue code for multiple error detection and correction.

LIST OF REFERENCES

1. Johnson, Barry W., Design and Analysis of Fault-Tolerant Digital Systems, Addison-Wesley Publishing Company, 1989.

2. Tsui, Frank F., LSI/VLSI Testability Design, McGraw-Hill Book Company, 1987.

3. Williams, Thomas W. and Parker, Kenneth P., "Design for testability-a survey," Proc. IEEE, vol. 71, pp. 98-112 January 1983.

4. Stressing, John, "Fault simulation and test generation - an overview," Computer-Aided Engineering Journal, vol. 6, no. 3, pp. 92-98, June 1989.

5. Williams, Jacob A., "Fault modeling in VLSI," VLSI Testing, T. W. Williams ed., pp. 1-27, Elsevier Science Publishers B. V., 1986.

6. Mangir, Tulin Erdim, "Sources of failures and yield improvement for VLSI and restructable interconnects for RVLSI and WSI: Part I - Sources of failures and yield improvements for VLSI," Proc. IEEE, Vol. 72, pp. 690-708, June 1984.

7. Davidson, John Carl, "Implementation of a design for testability strategy using Genesil Silicon Compiler," Master's Thesis, Naval Postgraduate School, Monterey, California, 1989.

8. Pooler, Brian L., "A methodology for producing and testing a Genesil Silicon Compiler designed VLSI chip which incorporates design for testability," Master's Thesis, Naval Postgraduate School, Monterey, California, 1990.

9. Wadsak, R. L., "Fault modeling and logic simulation of CMOS and MOS integrated circuits," Bell Systems Technical Journal, vol. 57, no. 5, pp. 1449-1475, May-June 1978.

10. Payne, D., "Silicon compilation in ASIC," Defense Computing, vol. 1, no. 6, pp. 38-40, November-December 1988.

11. Genesil System, Volume II, Parallel Data Module, Silicon Compiler Systems Corporation, San Jose, California, September 1988.

12. Funatsu, S., Wakatsuki, N., and Arima, T., "Test generation systems in Japan," Proceedings of the 12th Design Automation Conference, pp. 114-122, June 1975.

13. Johannsen, D. and Sabo, D., "Genesil Silicon Compilation and design for testability," 3rd International IEEE VLSI Multilevel Interconnection Conference, pp. 372-380, 1986.

14. Stewart, J. H., "Future testing of large LSI circuit cards," Digest of Papers of the 1977 Semiconductor Test Symposium, pp. 6-17, October 1977.

15. McCluskey, E. J., "Built-in self-test techniques," IEEE Design and Test, April 1985.

16. Rao, T. R. N., Error Coding for Arithmetic Processors, Academic Press, Inc., 1974.

17. Avizienis, A., "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," IEEE Transactions on Computers, vol. C-20, no. 11, pp. 1322-1331, November 1971.

18. Yeh, Raymond, T., Applied Computation Theory: Analysis, Design, Modeling, Prentice-Hall, Inc., 1976.

19. Kung, Chih-fu, "A pipelined implementation of notch filters using Genesil Silicon Compiler," Master's Thesis, Naval Postgraduate School, Monterey, California, 1990.

20. Genesil System, Volume I, Blocks, Silicon Compiler Systems Corporation, San Jose, California, September 1988.

21. Genesil System, Volume III, Parallel Data Module, Silicon Compiler Systems Corporation, San Jose, California, September 1988.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-6145	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Prof. Herschel H. Loomis Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	2
5.	Prof. Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	3
6.	Commander, Naval Research Laboratory ATTN: Lt. Brian Kosinski, Code 9110-52 4555 Overlook Ave., S.W. Washington, DC 20375	1
7.	Commander, Naval Research Laboratory ATTN: LCDR D. Barnes, Code 9120 4555 Overlook Ave., S.W. Washington, DC 20375	1
8.	Commander, Naval Research Laboratory ATTN: Dr. A. Ross, Code 9110-52 4555 Overlook Ave., S.W. Washington, DC 20375	1
9.	Commander, Operational Test and Evaluation Force ATTN: LT John E. Lawson, Code 721 Norfolk, Virginia 23511-5225	1

10. Commander, Naval Research Laboratory 1
ATTN: LT Kirk Harness, Code 9120
4555 Overlook Ave., S.W.
Washington, DC 20375